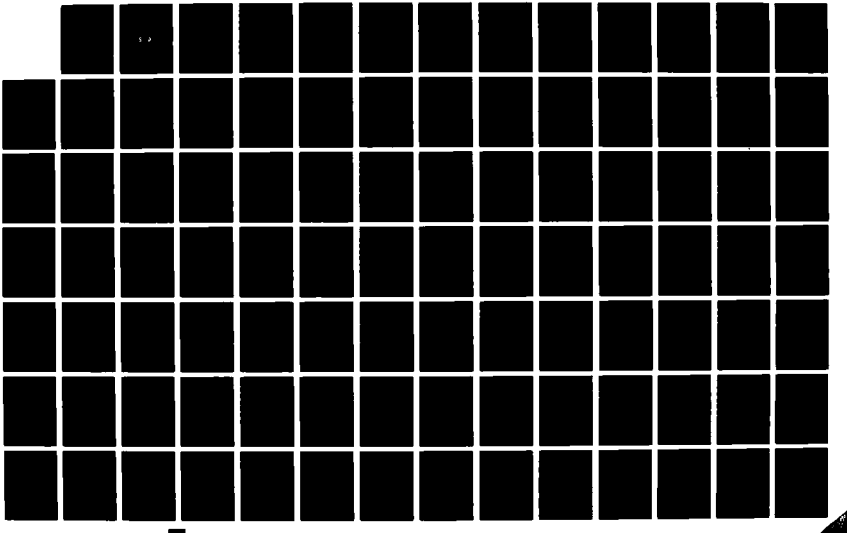
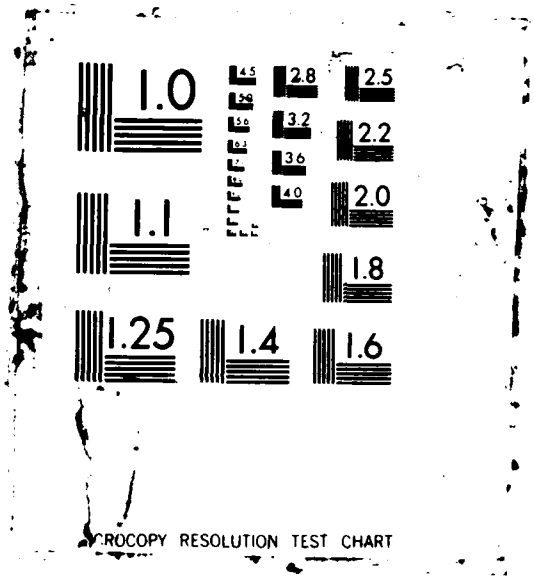
AD-A179 493 SEQUENTIALIZATION OF LOGIC PROGRAMS(U) STANFORD UNIV CA 1/2
DEPT OF COMPUTER SCIENCE R J TREITEL NOV 86
STAN-CS-86-1135 N00014-81-K-0303

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
CROCOPY RESOLUTION TEST CHART

November 1986

Report No. STAN-CS-86-1135

Sequentialization of Logic Programs

by

Richard Treitel

Department of Computer Science

Stanford University
Stanford, CA 94305

87

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A179 493 / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Sequentialization of Logic Programs	5. TYPE OF REPORT & PERIOD COVERED technical
	6. PERFORMING ORG. REPORT NUMBER STAN-CS-86-1135
7. AUTHOR(s) Richard Treitel	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Stanford University	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209	12. REPORT DATE November 1986
	13. NUMBER OF PAGES 179
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

logic programming, search, control of inference

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Please see reverse.

DD FORM 1 JAN 73 1473

19 KEY WORDS (Continued)

20 ABSTRACT (Continued)

Logical inference can be done in two directions: either forwards from facts already known to other facts that may be of interest, or backwards from goals whose answers are needed to subgoals whose answers may be available. Some programs and languages use one direction exclusively because it is clearly better (in the sense of computational efficiency) for their purposes. There are, however, problems that are better solved by using some rules for forwards inference and others backwards. In theorem-proving and artificial intelligence work over the past two decades, numerous systems have been developed which allow inferences to be drawn either forwards or backwards, and a number of heuristics have evolved for deciding which direction to use a rule in.

In this dissertation I attempt to put these decisions on a quantitative footing, by developing algorithms for estimating the computational cost of a set of directions for rules, and applying standard optimisation techniques to derive the best set of choices. In ascending order of difficulty, it is assumed first that no forward rule uses any facts deduced by a backward rule; then that directions can be chosen freely; and finally that each rule can be split up and used partly forwards, partly backwards. All of these problems, except for a highly constrained version of the first one, are shown to be NP-complete. For each of them, one or more optimisation techniques are given, with some comments on their efficiency. Techniques for "ameliorating" or "satisficing" the problem, namely returning a non-optimal solution that is good enough for the purpose at hand, are also discussed, since the effort of finding the exactly optimal solution may be excessive.

A related issue is the ordering of the terms in a rule, which can have a strong effect on the cost of using the rule. Algorithms for ordering terms optimally are known, but they rely on the direction of inference being fixed in advance, and they apply only to one rule considered in isolation. A more general algorithm is developed, and a way is shown to incorporate it into the choice of rule directions.

SEQUENTIALIZATION OF LOGIC PROGRAMS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Richard James Treitel

September 1986

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Michael R. Genesereth
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jeffrey D. Ullman

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Cordell C. Green

Approved for the University Committee on Graduate Studies:

Dean of Gradu te Studies & Research

Abstract

Logical inference can be done in two directions: either forwards from facts already known to other facts that may be of interest, or backwards from goals whose answers are needed to subgoals whose answers may be available. Some programs and languages use one direction exclusively because it is clearly better (in the sense of computational efficiency) for their purposes. There are, however, problems that are better solved by using some rules for forwards inference and others backwards. In theorem-proving and artificial intelligence work over the past two decades, numerous systems have been developed which allow inferences to be drawn either forwards or backwards, and a number of heuristics have evolved for deciding which direction to use a rule in.

In this dissertation I attempt to put these decisions on a quantitative footing, by developing algorithms for estimating the computational cost of a set of directions for rules, and applying standard optimisation techniques to derive the best set of choices. In ascending order of difficulty, it is assumed first that no forward rule uses any facts deduced by a backward rule; then that directions can be chosen freely; and finally that each rule can be split up and used partly forwards, partly backwards. All of these problems, except for a highly constrained version of the first one, are shown to be NP-complete. For each of them, one or more optimisation techniques are given, with some comments on their efficiency. Techniques for "ameliorating" or "satisficing" the problem, namely returning a non-optimal solution that is good enough for the purpose at hand, are also discussed, since the effort of finding the exactly optimal solution may be excessive.

A related issue is the ordering of the terms in a rule, which can have a strong effect on the cost of using the rule. Algorithms for ordering terms optimally are known, but they rely on the direction of inference being fixed in advance, and they apply only to one rule considered in isolation. A more general algorithm is developed, and a way is shown to incorporate it into the choice of rule directions.

Acknowledgements

This work was partially supported by the Office of Naval Research under contracts number N00014-81-K-0303 and N00014-81-K-0004, by the National Institutes of Health under grant number 5P41 RR 00785, and by the Defense Advanced Research Projects Agency under contract number N00039-86-C-0033.

Contents

List of Figures

Chapter 1

Introduction

1.1 Declarative Programming

1.1.1 Generations of Programming Languages

Successive generations of programming languages have advanced and gained acceptance by requiring less and less mental effort by the programmer in the composition of a program to perform a given task. This both increases the productivity of programmers and reduces the likelihood of programmer errors afflicting the code. Each such advance has been achieved by transferring responsibility for part of the programming task from human to machine, and large amounts of theoretical and practical work have been devoted to making this not only possible but economical.

For example, the change from programming in machine language to assembly code relieved humans of the burden of keeping track of which locations in main memory held a particular piece of code or data, and of having to remember binary operation codes. To achieve this, symbol tables had to be developed. It was soon seen that macro operations were a desirable adjunct to assembly code, and assemblers became more complicated as these were added.

Next, a large step forward was taken with the invention of FORTRAN, Algol, BCPL,

and other languages which allowed the human to ignore all details of the hardware on which the program would run. This was accompanied by great strides in parsing theory, register allocation algorithms, and optimisations of all kinds. Subsequently, languages using strong typing and data abstraction made it possible for machines to attend to certain aspects of program correctness, the price for this being that compilation took longer and was underpinned by yet more theory.

1.1.2 Logic Programming: an immature generation

Recently, declarative programming languages have emerged which claim to make life even easier by making it unnecessary to specify the sequence in which operations should be performed ("declarative" is used as an antonym for "procedural"). The programmer should only need to write down statements about what actions may be performed, and the machine will obey these. This would both remove a major source of unintentional mistakes and, at least in theory, make it easier to check that a program is in accordance with its specifications. However, this claim is not yet fulfilled by the commoner logic programming languages, such as Prolog [CM84].

Usually it is impossible to write a logic program so as to include all the desired inference steps while ruling out those that will not be productive. So it is likely that, before all the steps which would fulfill the program's purpose have been executed, some useless ones will be done. The sequence in which the steps are done is usually determined implicitly, often by the order in which rules appear in the program. Thus programmers can write inefficient programs just as easily as before, unless they have been made aware of the implicit sequencing that goes on behind their backs, and given some means of modifying it. The commonest example of this is a simple recursive definition:

$$\begin{aligned} P(x,y)\ \&\ Q(y,z) &\Rightarrow P(x,z) \\ R(x,y) &\Rightarrow P(x,y) \end{aligned}$$

Here, if the implementation both uses depth-first search (as most do) and tries the recursive rule before the base case, execution may fail to terminate.

Less drastic examples can be found where the wrong sequence of operations can lead to large increases in the running time of a program by causing many unproductive operations to be performed before the few that solve the problem. Warren [War81] and Smith [SG85] have pointed out that, if a rule has a conjunction as its antecedent, ordering of the conjuncts can have a strong effect on efficiency. It has also been observed that some declarative programs can potentially do very many repetitions of the same inference, unless arrangements are made to store its result for future use, a spectacular example being provided by the computation of the nth Fibonacci number:

$$\begin{aligned} Fib(n) &= Fib(n-1) + Fib(n-2) \\ Fib(0) &= 1 \\ Fib(1) &= 1 \end{aligned}$$

Here, if $Fib(n)$ is computed by straightforward backward chaining which does not store any intermediate results, the time taken to do so will be somewhat larger than $Fib(n)$ itself, if an addition can be performed in 1 unit of time. But using forward chaining, starting from the values of n for which $Fib(n)$ is already known and building up a table of values, we can reach the nth entry with $n-1$ arithmetic additions and $n-1$ new entries to the table.

Another example might be a genealogical system, aimed at finding out whether two people are or are not related by blood, namely

$$\begin{aligned} Ancestor(a, p1)\ \&\ Ancestor(a, p2) &\Rightarrow Related(p1, p2) \\ Parent(p, c) &\Rightarrow Ancestor(p, c) \\ Parent(p, x)\ \&\ Ancestor(x, d) &\Rightarrow Ancestor(p, d) \end{aligned}$$

Using backward inference only to determine whether two people were related could be very expensive, since for every ancestor a of $p1$ we would have to check whether a was an

ancestor of $p2$, and this could involve a very long search through all of a's descendants. On the other hand, using forward inference to find all pairs of related people would also be very expensive and would deduce many facts that might never be needed. Depending on how heavily the system was expected to be used, it might make sense to deduce all the $Ancestor(a, p)$ facts forward, and store them as the basis for backwards inference to answer questions of relatedness.

1.1.3 Growing up

Some logic programming languages, including MRS [Gen82] and the theorem-proving component of the Stanford Pascal Verifier [SVG79], recognise these issues and make explicit provision for the programmer to specify control information, such as where forward or backward reasoning should be used. The more mature languages are divided into "base-level" and "meta-level" parts, so that rules (or "knowledge") can be given without specifying how or when they should be used, and restrictions on their use can be added later. This approach and its benefits are explained in [Gen84]. Others, for no reason known to this author, advocate the opposite doctrine, namely that a piece of knowledge should be inseparably linked to the control knowledge about how to use it: it is impossible to be non-committal about how a piece of knowledge is to be used, and if it be desired to use the same piece of knowledge in two different roles, it must be represented twice. See, for example, [Hew75].

In order to achieve the expressed aim of logic programming, namely to free humans from the burden of specifying *how* things should be done so that they can concentrate on *what* should be done, control knowledge should be generated automatically. This certainly implies that it be separate from base-level knowledge, which is still assumed to be supplied by the human (or conceivably by a learning program). The work reported in this dissertation consists of a study of how certain control decisions can be taken automatically, insofar as they affect the cost of running a logic program.

Other criteria than cost are often influential on control decisions. One of these is non-monotonic inference: with forward inference, a particular fact may be absent from the database either because it cannot be proved, or because it will be deduced later on. These two circumstances will be indistinguishable to a piece of code that draw a conclusion from the fact's absence. Backward inference ensures that the attempt to determine whether a fact is true will cause all relevant deductions to be attempted: if the fact can be proved, it will be. So backward inference may be necessary to ensure correctness of the program. In planning, backward inference provides a handy way of accumulating the decisions made in the course of coming up with a plan, while forward inference can be used to obtain constraints on further decisions [FG85]. But it is rare that considerations such as these will entirely determine how the program ought to be run; there will usually be some room to tune it to reduce the cost of execution.

1.2 Kinds of control

Mechanisms for controlling a logic program, i.e. deciding which step to perform next, can be characterised along several different dimensions. I shall discuss the following possible dimensions:

- The time at which the decision is made.
- The time at which the decision is acted upon.
- The amount of computation needed to make the decision.
- The amount of domain-specific knowledge needed to make the decision.

1.2.1 Timing of control decisions

Since it is undecidable whether a given program will halt on a given input, there will always be a need for some kind of control over a program to be exercised at run-time, even if only

by the user's fingers (at least this can legitimately be called "digital" control). Further, termination is not the only property of a program cannot be correctly predicted before running it. Control decisions based on such properties must at least be subject to review at run-time, or else risk being completely wrong. In many cases, though, there is a real trade-off between the accuracy of a decision and the choice of making it in advance or when it is needed.

Experience with previous generations of programming languages does suggest that some control decisions are best made before the program is run, either by the programmer or by a compiler. The justification for this is that the same decision would have to be made at run-time anyway, and indeed it might have to be made many times as the program processes many pieces of data. Hence it can be better to make the decision once and remember it. Moreover, extra resources can be devoted to making the decision more carefully, if we expect to amortise this investment by applying it many times.

Against this we must set the potential difference in accuracy between control decisions made at run-time and at compile-time. A decision made at run-time will benefit from the availability of up-to-date information about whatever factors affect the decision, whereas one made earlier will have used some predictions of these factors, which might not be accurate. Besides, a decision that is applied many times will presumably be applied in many slightly different situations, and even if the details of these situations could have been accurately predicted, they would then have been averaged over before the decision was taken, or a compromise would be arrived at somehow. Such a compromise is likely to be non-optimal some of the time.

1.2.2 Timing of control actions

If a control decision has been made before the time when the actions to which it relates are performed, there can be different ways to implement the decision. This applies even to decisions made at run-time, if they are to be kept for future use in similar situations.

The simplest control strategies, such as depth-first backward chaining, can be thought of as decisions that have been implemented in the code of the inference mechanism. Generally, a control decision can be put into effect before it is needed by means of some data structure that represents the decision and can be used in a simple way by the code. Prolog uses the order of literals within a clause for this purpose.

More complicated strategies will usually be implemented by means of some form of control knowledge that is consulted at run-time; MRS is an example of a system where every deductive action is preceded by a control deduction that decides how to perform the base-level deduction. The advantage of this approach is obvious, in that control can be more flexible; its disadvantage, equally obvious, is slower execution. One can also conceive of intermediate points along this spectrum, where some processing of the control decision was done when it was made and some left until later.

1.2.3 Cost of control

The fact that uncontrolled inference is incapable of solving any but the simplest of toy problems needs no emphasis. A little effort invested in control will usually pay off handsomely. However, all automatic deduction still involves backtracking or some other kind of search, because perfect control (which would make search unnecessary by choosing the right operation to perform at all times) would be far more expensive computationally than search. Indeed, any implementation of perfect control would probably have to find the right operation by searching for it. So there must come a point at which the addition of stronger control will pay off negatively.

This may seem to set logic programming apart from "conventional" programming, but the difference is not as great as one might think: most sorting algorithms, though written in languages which provide explicit control, are in reality searches for a permutation that puts the data into order. The precision of mathematical results on the upper bound complexity of sorting obscures the fact that the performance of most sorting algorithms is strongly

affected by the original arrangement of the data. The uncertainty about the length of time it will actually take to sort a given data set is similar to the uncertainty about the performance of most searches.

It is customary to compare two sorting algorithms by the number of comparisons they perform, i.e. the cost of control, rather than by the cost of permuting the data into order, because the control cost generally dominates the other cost (permuting data items, or pointers to them, is very easy). But trying various permutations and seeing how close they come to the correct order is a ridiculous sorting algorithm, because there is only one right one. By contrast, in logic programming there may be many sequences of deductive steps that get the answer, and often the best way of finding out whether a particular sequence works is to try it. The underlying operations, namely deductions, are also computationally non-trivial. So the cost of deduction is itself significant and must be balanced against the cost of control.

1.2.4 Use of domain knowledge

Early work on controlling deduction was directed towards methods that made the control decisions when they were needed, very cheaply, and using no domain knowledge, with the aim of preserving generality. This virtually dictated that the criteria for performing an inference step be purely syntactic, and gave rise to well-known resolution strategies such as set-of-support, linear input, unit preference, and so on. While these strategies have satisfying mathematical properties (they are provably complete, never more costly than uncontrolled resolution, and often less costly), it was soon discovered that they were still too weak to make many interesting problems tractable. The correlation between the syntactic criteria and the actual usefulness of an inference step was slight. Other criteria, such as the length of a clause or the number of free variables in it, were inadequate for the same reason: they bore little relation to the content of a clause or to the difficulty or value of using it.

The search for stronger deductive methods took two main directions. One was towards

interactive "proof checkers", where a user would command the system to perform a step or sequence of steps, inspect the results, and give another command. Thus a high level of human intervention was involved, with decisions being made as they were needed, and using (presumably) plenty of domain-specific knowledge implicitly supplied by the user. As these proof checkers developed, more and more powerful commands were incorporated so that the ratio of machine work to human effort increased.

Opposite these systems were fully automatic "provers" requiring the user to distill the domain knowledge into some form of explicit "advice," "heuristics," "hints," or "program," which would be fed in along with the rules and data; the system would then run without any interaction. While the interactive systems needed continual human guidance, the automatic ones needed very careful programming. Deep knowledge of the problem domain was essential for writing these programs, and the conversion of this knowledge into machine-usable form was a black art. Papers advocating or describing automatic deductive systems are remarkably silent on how to advise or program them to solve specific problems; see, for example, [Ove76].

A more practical approach to control of inference was meanwhile being pursued by the relational database community. Though databases and inference had long been regarded as distinct areas of study, the interaction between them was recognised by several researchers in [GM78] and subsequent books and articles. There is an obvious transformation from relational database queries to logic programs, where relations become predicates, tuples of a relation become facts beginning with that predicate, definitions of intermediate relations or views become rules, and queries become goals. This transformation neglects many features of the database, such as the different access methods and indices that may be available for each relation, and the dependencies (functional, multivalued, etc.) that may be specified. So a query that has been optimised as a logic program may still need more work before it will run efficiently on a database. It also maps onto a kind of logic program unfamiliar to most logic programmers, where the rules are few and simple while the data are very

numerous indeed. But the very size of the data set makes good control of the deductive process imperative.

The control methods (called "query optimization algorithms") that were developed in response to this need did pay some attention to syntactic features of their input, but since they were intended to be usable on any database regardless of its contents, they could make no direct use of domain knowledge. Instead, they derived their power mainly from use of numerical information about the contents of the database, which enabled them to estimate the costs of using a particular control strategy on a query, and somehow search for the cheapest strategy, or at least a cheap one (the word "optimization" was misused here as it has often been elsewhere). Due to the potentially enormous cost of clumsily processing a query, it made sense to invest considerable effort in good control decisions, and this was done, especially by [SC75,Hal76,Ast76,WY76,CM77,Yao79,SAC*79].

The approach adopted in these database query processors of making control decisions based on database statistics has been called "semi-independent" by Smith [Smi85], as distinct from the completely domain-independent or -dependent approaches mentioned above. Here I prefer to think of it as "indirectly domain-dependent". The numbers used are not functions of the meanings of the symbols in the language that describes the domain, but they would generally be different for different domains.

Smith argues at some length that indirectly dependent control will not replace domain-dependent control. The depth of understanding of a domain which human experts can have is not within the grasp of current AI technology, and there is, in some cases, no substitute for control strategies based on this understanding. Still, when human expertise is unavailable or uneconomic, indirectly dependent control offers a mechanisable alternative that can strengthen a domain-independent strategy considerably, without sacrificing any generality. Completely domain-independent strategies, weak as they are, are better than nothing, and will remain indispensable as components of stronger strategies.

1.3 Scope and Aims of this Dissertation

The work described herein constitutes an extension of the attempt, begun in [Warn81] and [Smi85], to apply indirectly domain-dependent control to logic programs. In contrast to some of their work, control decisions are here assumed to be made and put into effect before running the program, with much computational effort devoted to them. The first objective of this investigation is to find out how much effort, relative to the size of the description of the problem being solved, is needed to find optimal members of the sets of strategies considered. Another aim is to describe the possibilities of reducing this effort by settling for a sub-optimal strategy. The risk of obtaining a non-optimal strategy given inaccurate inputs is also considered: robustness is always important in an optimiser.

When human programmers use computational cost as the criterion for making control choices, they do so on the basis of rough, intuitive estimates of the costs of different alternatives, and use simplified models of the deductive algorithms, which may not reflect the true cost of some operation. There will also be a tendency to pick "prominent" solutions, which reduce the cost of some part of the program that the programmer considers important, while the costs of other parts are perhaps increased. So an optimiser which uses approximate values for the costs, and runs some risk of arriving at a strategy that is not quite optimal, can still be useful, since it can plausibly be expected to do as well as a human would in most cases.

1.3.1 Control and Sequencing

The task of controlling a logic program can be divided into three subtasks. These are, first, the elimination of inference steps that are superfluous, because they can never contribute to the solution of the goal; second, the recognition of steps that are redundant with those that have already been done, and so can add nothing new; and third, the choice from among the remaining steps of one which is part of the most promising strategy for solving

the goal. There may not be a single "best" step to take at any given time, inasmuch as several steps may all be essential and the order in which they are taken may be of no importance.

Fully general control, in which all possible strategies for solving the goal are considered, has to be done this way, considering all legal inference steps, and can be very expensive. Such expense is usually avoided by devising a broad set of strategies which are fairly cheap to implement and have known properties in terms of superfluity, redundancy, and so on. Control, especially at compile-time, then reduces to choosing a strategy from among the set, and the wider the variety of strategies in the set, the more likely it becomes that one of them will be good for a particular problem. The sets of strategies that are considered here will be described in detail in Chapter 2. For now, suffice it to say that they will be built up out of well-known methods of forward and backward inference.

Superfluity is, of course, one of the major issues in choosing between forward and backward inference: forward inference can generate many superfluous facts, but may still be worth doing when compared to backward inference. The strategies we use do not forbid superfluous inference steps, but rather restrict them to those that are side-effects of useful ones. Regarding redundancy, we stick to inference strategies that are irredundant if the rules themselves are, so that the problem of redundancy is delegated to the programmer. The choice of which inference step to do next becomes a consequence of the strategy chosen, and is an important criterion for preferring one strategy over another.

1.4 Structure of this document

Chapter 2 describes the logic programs and deductive mechanism over which we will try to impose control, and explains the various sets of strategies that are examined in succeeding chapters. Chapter 3 gives the theoretical and practical basis of the estimates for computational cost that are used to select a strategy. An effort has been made to keep

the mathematical complexity to a minimum, except in Section 3.4, but this chapter is unavoidably both long and detailed. Since the optimisation is not very strongly coupled to the means by which the estimates are obtained, readers may skip lightly over this chapter; this will, however, be a disadvantage for those who wish to follow the discussion of errors in Section 8.1. The definitions in Section 3.1.1 are also important for understanding later chapters.

Chapters 4 to 7 are the main body of the dissertation. Each of them concerns itself with the problem of choosing the optimal (or a near-optimal) member of some set of strategies. Results about the complexity of the problem are proved, and supplemented by informal discussion of features which make the problem easy or hard. Based on this, algorithms are suggested and some observations about their likely performance are made.

Chapter 8 discusses two issues that bear more generally on such optimisations: the extent to which errors in the cost estimates lead to faulty optimisation, and the trade-off between the amounts of time devoted to optimisation and execution. It contains no significant theoretical results, but should be required reading for anyone who wants to make money by optimising logic programs. Chapter 9 summarises the achievements of this piece of research and offers some directions for future work.

Chapter 2

Definition of Problems

In this chapter we explain what optimisation problems are dealt with in the rest of the dissertation, and what assumptions are made about them. Some of the terminology and notation used is introduced here, and some in the individual chapters where it is used.

2.1 The Programs

A logic program, for the purposes of this work, consists of a set F of facts (ground atomic formulae of the predicate calculus), a set R of rules (sentences of the predicate calculus having implicational force), and a set G of goals (conjunctions or atomic formulae). Rules and goals may contain variables, which are assumed to be universally quantified in the case of rules and existentially quantified in the case of goals. No function symbols appear in R or G. F and R are assumed to be finite; G need not be finite. The purpose of the system is to solve each goal from G using the facts F and the rules R, and, in the case of goals containing variables, to find all the sets of bindings for these variables which make the goal true. The case where only a single answer is sought could be dealt with by adjustments to the cost estimation algorithms described in Chapter 3.

Clauses from R will be required to be Horn clauses, i.e. have exactly one positive literal.

The facts are also assumed to be positive units, and the goals to contain only negated literals. This restriction on the signs of literals is not strictly necessary, and can be replaced with the requirements that each rule clause have a distinguished *output* literal, of either sign, and that no deductive steps will be done that unify two input literals or two output literals (from the same or different clauses). An *input* literal is one that is not an output literal. Each fact consists of an output literal, and a goal has only input literals. Note that this is only a relaxation of the restriction on where $\neg$ signs can appear, and does not imply any commitment to "negation as failure", or indeed any increase in power over standard Horn clauses. Those who are more comfortable with Horn clauses are free to substitute "positive" for "output" and "negated" for "input" where appropriate in the rest of this dissertation. The "inputs" of a rule will be understood to be the facts that can unify with its input literals.

Define a directed graph, called the *rule graph*, whose nodes are the members of R and which has an arc from a rule r to a rule s iff r's output literal is unifiable with one of s's input literals. This will be described by saying that s is a *successor* of r, and r a *predecessor* of s. The rule graph will usually be required to be acyclic, since many of the techniques and algorithms given will break down when there are cycles in it. Techniques for dealing with cycles, i.e. recursive rules or sets of mutually recursive rules, are being investigated by many researchers [NH80,MS81,HN84,Ull84]. There will also be times when we add F and G to the rule graph, in the obvious places: a fact from F is the predecessor of those rules whose input literals unify with it, and a goal from G is the successor of those rules whose output literals unify with it. The rule graph does not contain nodes for individual members of F and G, but for sets of facts or goals that match some pattern; these patterns, rather than the individual facts or goals, are unified with literals from rules when constructing the rule graph. Aside from this, the rule graph is similar to the connection graphs of [Kow75], but it is used only as a static structure: no deductions or other transformations of it are contemplated. In Figure 1, we have put the facts at the bottom and the goals at the top,

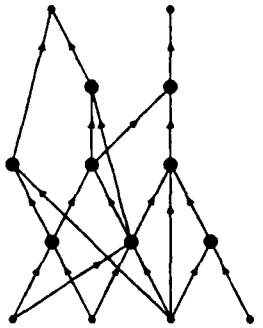


Figure 1: A moderately tangled rule graph

and we shall adhere to this convention henceforth.

It may be that some of the facts in F are not found by lookup, but are proved by calling an "attached" procedure, written in a conventional programming language. This is often done for simple predicates, such as arithmetic ones, and sometimes for more complex ones whose theory is sufficiently well understood that they can be implemented efficiently in this way. See [Sho84,NO79] for discussions of how this can best be done.

2.2 The Deductive Mechanism

Lock resolution [Boy71] is a restricted version of standard binary resolution [Rob65], the restriction being that the complementary literals which are resolved away must each be the first in their respective clauses, where "first" is defined relative to the ordering induced by a global labelling of literals with integers; substitution instances of a literal inherit its label, or "index". Thus the number of possible resolutions on a given clause set is substantially reduced, compared with unrestricted resolution. Lock resolution is, however, still complete if no other restrictions are added.

Here, a further restriction is imposed, namely that the ordering on the literals should correspond to the textual order in which they appear in a clause. Thus, only the literal appearing first, or leftmost, in a clause can be resolved upon. We call this restricted

form of resolution *LL-resolution*. When generating a resolvent, the remaining literals from the parent clause whose first literal was positive (output) are put at the beginning of the resolvent; another way to say this is that they are put in the place of the negative literal that was resolved away from the other parent. We call this "negative replacement", and denote the combination of it and LL-resolution by *LLNR-resolution*. It is, of course, not always the case that LLNR-resolution will yield a resolvent whose textual order still corresponds to the order induced by the indices of the literals in the parent clauses. But in the case of a recursion-free rule set we can ensure that the two orders will always agree, by the simple device of assigning the indices so that each literal in any clause (to be exact, in any rule or fact) has an index lower than any literal in any successor of the clause. Thus LLNR-resolution becomes a special case of lock resolution in such rule sets.

2.2.1 Properties of LLNR

Like lock resolution, LLNR-resolution is complete on arbitrary sets of clauses; the proof is deferred to another paper. We shall see that on Horn clause sets such as we have been discussing, it can give behaviour very similar to that of pure Prolog.

One other property of LLNR-resolution on Horn clause sets is of interest, namely that in any resolvent beginning with a positive (output) literal, the other literals will all be substitution instances of literals which originally appeared after this literal in the same clause as it. This is in fact proved by way of a more general result, namely that any literal appearing after an output literal in a clause generated by LLNR-resolution must have come from the same clause as it.

The proof is by induction on the number of LLNR resolution steps done. Certainly the property is true before any resolution steps have been done. Now suppose that a Horn clause set has this property after some number of LLNR steps (it is easy to prove that resolution on Horn clauses can never generate a non-Horn clause, i.e. one containing more than one output literal). Then if any LLNR resolvent contains an output literal, this literal

must have come from the parent clause that began with an input literal, since the other parent contained only one output literal, which was its first literal and so was resolved away. The literals from the parent that began with an output literal all appear on the left of the resolvent, due to the negative replacement rule, and so none of them can appear after any literal from the other parent. So, if this other parent included an output literal, the literals appearing after it in the resolvent will just be substitution instances of those that appeared after it in the parent, and the property continues to hold.

2.2.2 The agenda

The implementation of LLNR-resolution that we shall consider includes a stack or *agenda* of clauses to be resolved against. To use a new clause, we add it to the top of the agenda. Then repeatedly pop the top clause off the agenda, store it in the database, resolve it against all possible clauses in the database, and add the resolvents to the agenda. When the agenda is empty, we have deduced all the consequences of the new clause. This amounts to adding the set-of-support restriction to LLNR-resolution. It is known [CL73] that the combination of lock resolution with set-of-support is in general not complete, and this is also true of LLNR-resolution. To ensure completeness, we must augment the traditional set of support (which consists of just the goal or goals) with some facts and/or rules, as discussed below.

When the clauses entered by the user are all Horn, and the rule clauses have their positive (or output) literal at one end or the other, LLNR-resolution begins to look very much like traditional forward or backward chaining. Consider the first rule from the above example, which can be written as the clause

$$Related(p1, p2) \neg Ancestor(a, p1) \neg Ancestor(a, p2)$$

With this, a goal of the form $\neg Related(x, y)$ will resolve immediately, giving a clause like

$$\neg Ancestor(a, p1) \neg Ancestor(a, p2)$$

amounting to a conjunction of subgoals, just as backward chaining generates subgoals from a goal. The rule can also be written as

$$\neg Ancestor(a, p1) \neg Ancestor(a, p2) Related(p1, p2)$$

and this can be resolved against two $Ancestor(a, p)$ facts one after the other, giving a fact of the form $Related(p1, p2)$ which is a consequence of the *Ancestor* facts, as would happen under forward chaining.

Putting the output literal in the middle can also be of use. A description of digital circuitry may contain rules like

$$Type(x, NotGate) \,\&\, Input(x, False) \Rightarrow Output(x, True)$$

If this is written as the clause

$$\neg Type(x, NotGate) Output(x, True) \neg Input(x, False)$$

then it can resolve against a fact like

$$Type(G87, NotGate)$$

to give

$$Output(G87, True) \neg Input(G87, False)$$

which is shorter than the original rule and may be cheaper to use for inferring the behaviour of $G87$ backwards.

Since a goal from G consists only of input literals, it can resolve only against clauses having their output literals at the beginning. The resolvents will then also consist entirely of input literals. This has the important consequence that such clauses will never resolve against each other, and so can be deleted once they have been resolved against the rules. But a rule whose output literal is not its first literal will not be used at all if the only clauses that can be put on the agenda are those consisting entirely of input literals. There are two

ways to ensure completeness in the presence of such rules: either put facts (output unit clauses) on the agenda when they are added to the database, assuming that the rules are already present, or put the rule itself on the agenda once the facts that could unify with its input literals are in the database.

The first of these is traditionally used, since the purpose of forwards inference is seen as making sure that a system is always aware of the consequences of facts that have been fed to it. It is sufficient for completeness that all facts unifiable with the first literal of any rule should be either in F or deduced forwards by other rules, since either of these ensures that they will appear on the agenda. The clauses obtained by resolving such facts against the rule will then appear on the agenda, so that facts unifying with the second and subsequent input literals must either appear on the agenda after the first fact does, or else be in the database (or provable by backward inference using facts already in the database) before the time such resolvents are added to the agenda. This requirement will be satisfied provided that all clauses containing a substitution instance of any output literal are kept in the database, because out of the clause and the fact that unifies with its first literal, whichever appears second will be on the agenda when the other is in the database, and so the resolution will occur.

However, keeping all of these clauses in the database can be expensive, since there may be very many of them, especially if some input literal of a forwards rule is solved by backwards deduction rather than lookup. An alternative is available when the rule graph is acyclic and when all the facts in F are guaranted to be present before a specified time. Acyclicity of the rule graph allows us to use a topological sort [Knu73, pp.258-265] to order the rules so that every rule appears after all of its predecessors. Then, once all the facts are present, we can just put the rules on the agenda in this order, leaving out rules whose output literal appears first. This ensures that, by the time any rule starts its work, everything on which it could possibly depend has already been deduced, so there is no need to store any of the generated clauses in the database except those that begin with an output literal.

2.3 The Cost Estimates

The cost estimation algorithm described in Chapter 3 takes as its input the whole of R, enough information about F to enable the number of answers to each potential goal or subgoal to be estimated, and enough information about G to make it possible to estimate the number of times a goal would need to be solved backwards if its answers were not already stored. The nature of this information will be described more precisely in Chapter 3. In database terminology, R corresponds to the "intensional database" or to a set of view definitions, F to a description of the contents of the "extensional database" or to some set of indices, and G to a "query model" describing the frequencies of possible queries. The cost estimates are based on estimates of the numbers of facts and subgoals obtained, which are arrived at by means of a simulation or symbolic execution of the program. The principle underlying the simulation is similar to that used for estimating the sizes of derived relations in database query optimisation work; see, for example, [SAC*79].

In the case where G is finite, the total cost to be minimised can be simply the sum of the estimated cost of forward inference and that of proving the goals in G. If G is not finite, then clearly the time taken in forward inference cannot be added to the total time taken in answering questions, but it can be considered as an investment in quicker answers, whose payback time can be calculated. In the case of a database, neither F nor G is finite, and what may be of interest is the cost of computing the sets of new facts that are entailed by each of an indefinitely long series of periodic updates, plus the cost of answering the (finitely many) questions asked before the next update. For a program that has to answer goals over a long period of time, the cost of the space used to store the clauses deduced forwards is also important, so a daily rental charge for disk space should be added to the cost of the CPU cycles consumed by the deductions. For an ephemeral program, the cost of space is less of a consideration; we assume, in fact, that space in main memory and on the swapping disk are free.

2.4 The Optimisations

The control decisions that we consider can all be expressed, under the LLNR-resolution mechanism, as changes to the order of literals in rules. No migration of literals between rules, or formation of new rules or new literals, is considered; neither is re-ordering of rules, since this is pointless when all the answers to a goal are wanted. The re-ordering of literals is further split up into the placement of the output literal and the permuting of the input literals. We shall consider successively broader classes of strategies obtained by allowing more and more freedom in these two directions. In all cases the objective will be to find the strategy giving the lowest value of the estimated total cost, or at least a strategy giving an acceptably low value. For discussion of what may be "acceptable" under various circumstances, see Section 8.2.

2.4.1 Position of output literal

We have already seen that the transition from forwards to backwards inference can be effected by moving the output literal from one end of the rule to the other. The two easiest optimisation problems studied will concern only this kind of change to a rule. To help describe strategies, we define an indicator variable $v(r)$ for each node r of the rule graph, with a value of 1 to denote using the rule forwards, and 0 for backwards. When ordering of input literals is not being considered, a strategy will be just a set of values for the $v(r)$. If some of the $v(r)$ are left unspecified, it is a *partial strategy*. We can also describe a complete strategy by the set R_f of rules that are used forwards.

To start with, we shall impose the additional restriction that all predecessors of a forwards rule be used forwards themselves, also implying that if a rule is used backwards, so are all its successors. This will be called *coherence*, i.e. a coherent strategy will be one in which R_f is closed under taking predecessors. Some deductive systems do in fact enforce coherence, and some others have a bias towards it. Even though there are usually many

possible coherent strategies for a given rule set R, coherence turns out to be a very strong condition on strategies, strong enough that the optimisation problem can be solved by a polynomial-time algorithm in some cases, and fairly quickly in most others.

By abandoning just the coherence restriction, we expand the set of strategies enough to take in the behaviours of many logic programming systems. Some of these do not permit entirely arbitrary incoherent strategies, or rather will give incomplete inference on them, because if forwards inference is done by putting newly entered or deduced facts on the agenda, it is necessary for each forwards rule to have at least one forwards predecessor. It may be that all rules whose output literals unify with the first input literal of a forward rule are required to be forwards. Such constraints, if present, do not significantly alter the nature of the optimisation problem, though they make the search for an optimal incoherent strategy a little easier. Finding this strategy is in general an NP-complete problem.

For full generality, we can allow the output literal to be anywhere in the rule. Rules used this way will be called "hybrid," since they are partly forwards and partly backwards. By extension, strategies that incorporate such rules will also be called "hybrid." Intuitively, a clause with its output literal in the middle is less like a rule than a generator of rules. The input literals before its output literal can be thought of as premises, and when these have been satisfied, a new backwards rule is generated instead of another fact. Thus the rules available for backwards inference will depend on the facts fed to the system. Finding the best hybrid strategy is harder than in the incoherent case with only pure-bred rules.

2.4.2 Inclusion of literal ordering

As was remarked in Chapter 1, the order of the input literals in a rule can be very important for the efficiency of its use. This applies equally to database queries, where much effort is devoted to getting operations in the most economical order. Under our deductive mechanism, the order in which input literals are proved is just the order of their appearance in the rule, so it can easily be controlled at "compile-time."

The problem of optimal input literal ordering will be examined for the coherent, incoherent, and hybrid cases, in order of increasing difficulty (as one might expect). We shall see that the optimal order for a rule considered in isolation is affected by the expenses of obtaining its inputs. Since these are in turn affect by the directions of the rule's predecessors, input literal ordering cannot be independent of output literal placement. We shall also see that rule directions are affected by rule costs, which are affected by literal ordering, so the optimal strategy can be obtained only by an algorithm that considers both.

2.5 Related Work

2.5.1 Logic programming

From the very beginnings of work on automated reasoning, it has been widely recognised that good control, i.e. avoidance of unproductive inferences, is essential if non-trivial problems are to be solved within the performance of current hardware. This fact is invariant with respect to advances in hardware technology, since these advances are outpaced by the increased complexity of the problems to which a solution is desired. In consequence, much has been written about how to constrain the inference process tightly enough to find answers quickly without constraining it so tightly that it misses some of the answers. See, for example, [Nil80, Ch. 5] and [CL73, Ch. 6,7]. However, all of the control methods mentioned in the standard texts are either of the weak domain-independent kind or else stronger but very domain-dependent, as described in Section 1.2. The advice given by [WOLB84] to those who would dabble in automated reasoning splits up into two kinds: syntactic strategies such as "Rely on shorter clauses ..." and "Use hyperresolution ...", and advice to use "... your knowledge and intuition about how the problem ... might be solved."

Kowalski [Kow75] makes a nod towards the idea of trying to estimate the branchiness of a deduction tree by using numerical information about the clauses whose existence is already known, though this has been overshadowed by the more important content of the

same article. The approach he suggests does not seem to have been followed up, though he claims that an upper bound can be obtained in time proportional to the number of levels looked ahead. Some heuristics for efficient use of rules are given in [Nil80, Ch. 6], depending on characteristics like branching factor; these, though, are supposed to be for the benefit of the human programmer.

The first implementation of automatically made control decisions appears to be in the CHAT system [War81], which tries to optimise a single conjunctive query. In our language, this corresponds to the case where there is only one rule, one goal, and backwards inference; moreover, the "optimisation" is done by repeatedly choosing the currently cheapest conjunct as the one to be done next, and this does not always give the optimal execution. Smith [Smi85] shows how to find the optimal ordering for a conjunction, under the assumption that execution costs can be predicted exactly. He also applies the indirectly domain-dependent control approach to three other problems, including the one of deciding how to order rules optimally when only a single answer to the goal is needed (so that it can probably be obtained without even trying the rules that come last). He is now investigating the combination of conjunct ordering and rule ordering for backwards inference.

2.5.2 Databases

The relational model for databases [Cod70] has strong advantages over the hierarchical and network models, but since the query languages used in relational database systems are normally higher-level than in others, the gap between the form of a query that is most natural to the user and that which is quickest in execution can be wide. Much of the work on relational databases has been aimed at optimising queries so that they can be answered in times that are competitive with those that would be achieved under the older models. While there is considerable scope for optimising the individual relational operations (selection, projection, join etc.) by various means, it has also been clear from the outset that careful choice of which operations to do and in what order to do them is vital. Emphasis on

this has increased with the trend towards equipping relational databases with logical front ends that pre-process a query into a set of primitive database operations [GM78,GMN80]. For an extensive survey of database query optimisation techniques, see [JK84].

Query optimisation

Most of the query optimisation tactics reviewed in [Ull82] can be regarded as examples of re-ordering: doing selections and projections early, re-arranging joins. Shifting of selections can also be regarded as avoidance of forward inference, inasmuch as a join (or other operation) done on relations that have not yet been operated on using any term from the query/goal is like a piece of forward inference done before the goal was known; doing selections before joins rather than after them reduces the extent to which such inference is done. Thus there is a strong relationship between query optimisation and the work presented here. Two major differences should be noted, though.

First, the algorithms that explicitly re-order queries, such as those in [WY76,SAC*79], admit to using heuristics to speed up the process, even at the expense of optimality of the final ordering. While the problem is combinatorially hard in general, the optimal ordering is frequently obtainable at low cost, and the heuristics do not even guarantee a solution within any given distance of optimality, though they perform well on most practical cases. In this work, algorithms wil be given that can find the optimal ordering quickly in most cases, and can usually be made to degrade gracefully in hard ones.

Secondly, the overwhelming majority of query optimisers consider each query as an isolated event. This applies even to the feature mentioned in [Cha81] whereby a query can be compiled into machine language if it is expected to be called, perhaps many times, from a conventional program. Thus, all temporary relations generated during query execution are assumed to be thrown away at once, which can be wasteful. The importance of retaining some such temporaries at least until the current query has been finished with was recognised in [Hal76,GM80], who however only considered relations that are exactly repeated in the

processing of the query. Jarke [Jar85] advanced to consider retaining a temporary if it were a superset of another that would be needed later during the processing of a set of queries. Kim [Kim85] proposes some heuristics for grouping a set of queries so that queries in the same group can make best use of each other's temporaries. Neither of these papers pay much attention to the complications attendant on having an inferential component to the database. They also ignore the potential value of materialising a derived relation which is larger than would be needed for any one query, but has parts that will be useful to many different queries. This was recognised by Schkolnick and Sorenson [SS81], who propose maintaining some of these relations on disk, namely those obtained by a join. However, the hill-climbing algorithm they propose for choosing the set of joins to be materialised is non-optimal.

Index selection

A separate part of relational database research has always dealt with entire relations and with the set of all queries expected to be asked. This is index selection. An index is a secondary data structure which allows fast access to all the tuples of a relation having a particular value or values at a fixed subset of the relation's attributes. Most indexes use only one attribute, so for example a relation connecting employees and departments could be indexed on the department attribute, making it easy to find all the employees who work in the Marketing department, but giving no help in finding all the departments where Jones works. The cost of providing disk space for an index can be a substantial fraction of the cost of storing the relation it indexes; updating the index to be consistent with the relation is also likely to add subtantially to the cost of update to the relation. Thus database systems must refrain from generating an index into some relation on some attribute unless there is clear evidence that it will be helpful to do so, i.e. if there will be many queries against that relation where the value of the indexed attribute is known. Generating and maintaining an index can be viewed as an investment in quicker answers, just like forward inference.

Merely finding the optimal set of indices for a single relation when query frequencies are known is hard [Com78], and in any case does not correspond to any control decision for a logic program. However, Roussopoulos [Rou82a] has examined the issue of maintaining indices on derived relations. Inference is explicitly included, and a search algorithm is given for finding the optimal set of indices if the frequencies of accessing the derived relations are known (a method for estimating these is expounded in [Rou82b]). We shall see, however, that not only is Roussopoulos' search algorithm inefficient, but that the quantities he uses to compute the optimal set of indices can change once the new set of indices is in use, due to the different decisions that a query optimiser would make if it knew that the indices existed. Thus, his optimisation may invalidate itself as soon as it has been done. This difficulty is recognised in [FST86]. The present work shows how to allow for the disturbing effect of query optimisation and find an optimal set of derived data to be maintained that is stable with respect to it.

2.5.3 Differences

While the connection between logic programming and relational databases is important and likely to grow more so, there remain some differences in assumptions and outlook between logic programmers and database workers, which not only cause different sets of problems to be explored, but also lead to different answers being held up as best or right when both sides look at the same problem. These differences are all or nearly all based on the disjoint classes of applications considered.

In database research, it is assumed that there is a large amount of data and relatively few rules, so that disk page fetches and storage limitations are more important than CPU cycles. This assumption is explicitly denied in [Cha81] but is used widely elsewhere. In its extreme form, it leads to the doctrine that a query is optimised once the number of join operations in it has been reduced to a minimum. Any logic programmer would laugh at this. The above-mentioned need for indexing to be done parsimoniously is also less strongly

felt in logic programming, and this divergence leads to disagreement about the true cost of retrieval operations.

Logic programming, on the other hand, is accustomed to ignoring questions of storage almost entirely, and regarding retrieval as just another inference operation. Cavalier attitudes toward disk access are believed to have been responsible for "inexplicable" inefficiency in at least one large expert system, which turned out to be doing an excessive amount of paging. In the future, as logic programming applications grow larger, or are run on smaller machines, database-like analysis of their efficiency will doubtless be more important than it is now. Meanwhile, this work unashamedly espouses the logic programmer's point of view.

Chapter 3

Estimating Costs of Deduction

We estimate the costs of running LLNR resolution on a set of rules, facts, and goals by means of a simulation, in which we represent each set of similar clauses that will arise during the computation by a clause, called the set's *pattern*, and a number, namely the expected number of instances of that pattern that will be generated. The sets F and G must be represented this way, since we do not expect to know exactly what facts or goals will be in them. If we regard a clause in R as having itself as pattern and the number 1.0, then we see that the basic step needed for estimating costs is to simulate resolutions between pairs of such clause sets using this representation.

The simulator accepts the sets of clauses from F, R, and G as input, and runs LLNR resolution on them, using a special "simulated unification" which is described in Section 3.1, obtaining descriptions (in terms of pattern and set size) of all the sets of clauses that can be generated. It has an agenda like the one described in Section 2.2, so that the effects of putting clauses on the agenda in different orders can be simulated, although this is not important at present. However, it is not possible to simulate accurately the result of putting some of the clauses from a set on the agenda at one time and the rest at a later time, unless two sets having the same pattern are created and added to the agenda at different simulated times.

We can supply the simulator with knowledge about how long each elementary operation (unification, substitution, and so on) will take, to arrive at actual time estimates for a logic program, as described in Section 3.2. Estimates of space needed are likewise obtained from the estimated numbers of clauses, the amount of space needed to store a clause of a given size, and the policy that determines which clauses will be stored. This policy is assumed to be fixed by the deductive software used, rather than being one of the parameters for optimisation. The time and space estimates can then be combined in various ways, as described in Chapter 2, to give a cost value. It is useful to decompose this simulated cost into a sum of rule costs; the difficulties involved in doing this are discussed in Section 3.3. Section 3.4 discusses the computational cost of obtaining all these estimates, which is an important component of the entire cost of optimisation.

3.1 The Number of Resolvents

Here we describe how to estimate the number of resolvents generated in each set, given estimates for the sizes of the sets in F and G. The simulator also needs to know, for each variable in any clause of R, the size of the domain of values over which that variable will range (or, in database terminology, the size of the projections of a relation onto each of its attributes). This is important for two reasons. First, some of the equations below are couched in terms of the probability of a typical instance of some clause pattern being generated, so that in order to convert this into a cost estimate, we need to multiply by the number of potential instances, which depends on these domain sizes. Domain sizes also affect the probability that two variables will be bound to different values, which in turn affects the chance that a unification will be successful; however, this probability may be known independently. Note that "domain" here refers to the set of values expected to be encountered during a particular run of the program, rather than to a set of theoretically possible values.

3.1.1 Notation

At this point it is useful to distinguish the set of variables in the first literal of a pattern which will have had constants substituted for them at the time when unification is attempted. We call these "bound variables" of the pattern, meaning that the simulation must know that they will be bound at run-time to constants whose values are not known yet. The other variables in the pattern will be referred to as its "free variables", or "variables" if there is no ambiguity. In what follows, we shall use single lower-case letters to denote free variables in patterns, and single upper-case letters to denote bound variables. Patterns of the same syntactic form but with differing sets of bound and free variables are treated as distinct, and a clause is said to match a pattern only if it has the same set of free variables as the pattern. Clauses obtained from a pattern by turning different sets of its free variables into bound ones will be called different *modes* of the pattern, following the terminology used by Warren in [War77], since bound variables correspond to input parameters and free variables to output ones (but the use of "input" and "output" here has no connection with input and output literals as defined previously). This is actually an extension of Warren's terminology, since he considers modes to apply only to literals; we shall usually consider modes of literals.

Under our assumption that all actual facts will be ground, a fact pattern cannot contain any free variables. It can contain both constants and bound variables. The pattern

$$StateOf(X, U.S.A.)$$

with the number 20 attached, tells the simulator that the actual facts will include the names of twenty out of the fifty United States, but gives no information about which ones. Rules will normally have no bound variables, since we assume that R is described completely. Goal patterns may have free and bound variables and constants. For example, the goal pattern

$$\neg Bought(I, 500, b)$$

with the number 1, means that the system will be given the name of a single item and asked

to return everyone who bought 500 units of it.

Define the *expense* of a pattern p under some strategy to be the total cost of all resolutions that could be done by putting a clause c matching p on the agenda and executing the strategy, but refusing to put on the agenda any clause shorter than c (i.e. containing fewer literals). Given the LL resolution restriction, the only clauses shorter than c that could be generated starting from c would be those that were substitution instances of c with its first literal, or some of its initial literals, stripped off. The expense can therefore be thought of as the price of solving the [sub]goal represented by c's first literal, which should be an input literal. At times, expense will be talked about as though it were a property of the first literal only, though as we shall see in Section 3.2, it can be affected by the length of the clause involved. The number of clauses shorter than c obtained from putting one copy of c on the agenda will be the number of answers to c, or to its first literal, and the expected number of answers to p will be the same number as estimated by the simulation. Different modes of the same literal, or pattern, will usually have different expenses, and can also have different ratios of expense to number of answers.

For a rule r let $e_f(r)$ be the estimated cost of using it forwards, net of the costs of using its predecessors to deduce its inputs, and $e_b(r)$ the estimated cost of using it backwards, again net of costs for its predecessors. $E_f(r)$ depends on the structure of R and on the number and distribution of facts in that part of F from which r's inputs are obtained, and $e_b(r)$ can depend both on these and on that part of G from which clauses unifying with r's output literal are obtained. Moreover, $e_b(r)$ will depend on whether any of r's successors (or their successors, etc.) are used forwards, since this can affect the number of clauses unifying with r's output literal, as explained more fully in Section 3.3. But when the input literals of r's successors may be permuted, it becomes unrealistic to even try to estimate $e_b(r)$, and we are forced to assess the cost of the backwards part of a strategy in terms of expenses.

3.1.2 Simulated unification

Now clearly the pattern of a set of resolvents will be just the result of resolving the patterns of the two parent sets. However, when a pattern that has bound variables is unified with another pattern, this represents some unifications at run-time in which constants will have been substituted for these variables, and the simulation must take account of this by treating bound variables differently from the way in which variables are treated in ordinary unification.

When a bound variable in a pattern is unified with a constant in another pattern, this represents run-time unification of some constant whose value is not known at simulation time with another constant whose value is known. So the value that will appear in the unified term is knowable at simulation time: it will be the second constant, and this is what must appear in the result of the simulated unification.

By contrast, when a bound variable is unified with a free variable, this represents unification of some constant with a variable. The result will be the same constant, but since the simulation does not yet know its value, only the name of the bound variable can be used to denote it, and so this name must appear in the simulated result. Thus we see that a bound variable, in simulated unification, acts like a constant when unified with a variable, but like a variable when unified with a constant.

There is a third case, in which two bound variables are unified. This represents unification of two constants, neither of which is yet known. Clearly the only possible result here is that if the two constants are equal, their common value will appear in the result; this is simulated by choosing one of the bound variables and substituting it for the other one. So a bound variable acts like a variable under simulated unification with another bound variable.

Unifications of two constants at run-time will fail unless the constants are in fact equal. So when a bound variable is unified with another bound variable or with a constant, the simulator should produce not only a unifying substitution but also a number, the probability

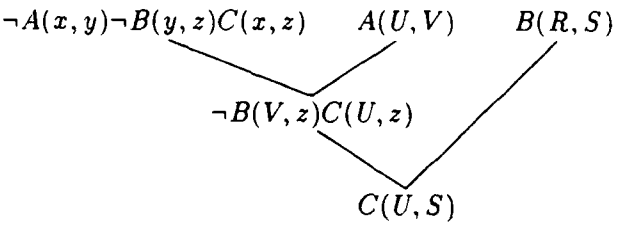


Figure 2: Some simulated resolutions

that unification will succeed.

To make this more concrete, consider a rule $\neg A(x,y)\neg B(y,z)C(x,z)$ and fact patterns $A(U,V)$ and $B(R,S)$, where the variables in the rule are free and those in the fact patterns are bound. Resolving the first fact pattern against the rule yields the clause pattern $\neg B(V,z)C(U,z)$. The reason why x and y have been replaced by bound variables is that, at run-time, constants from the $A(U,V)$ facts would be unified with these variables, and would give rise to constants in the resolvent. Now resolving $B(R,S)$ against this clause, we will get $C(U,S)$ where z has become a bound variable in the same way; moreover, at run-time this unification will only succeed in the cases where the actual value of R from $B(R,S)$ is the same as that of V in the instance of $A(U,V)$ that gave rise to the intermediate clause.

If the second fact pattern had been $B(1,2)$, this would tell the simulator that this fact would be present, so it should deduce that resolvents like $C(U,2)$ would be generated from facts like $A(U,1)$.

3.1.3 Probability of unification

In the absence of specific information, we can estimate the probability of successful unification between a bound variable and a constant (or another bound variable) by assuming that all values in the domain of the bound variable are equally likely to occur. This is called the *equal frequency assumption*: no constant appears more often than another of the same

type. The probability of such a unification succeeding is, of course, the reciprocal of the number of possible values in the domain, under the equal frequency assumption. Even for domains that are in principle infinite, such as the integers, any one set of input data will contain finitely many values. Then any finite computation will be able to generate only finitely many more values, and the probability of two values being equal is finite.

In general, two literals being unified by the simulator may contain several pairs of constants or bound variables that must be equal for unification to succeed. We make an *argument independence assumption*, under which the event of one pair being equal is independent of other such events, so the probabilities can be multiplied.

It may be argued that probability distributions other than the uniform one should be allowed in order to give a more faithful description of the input. However, there is no convenient way to represent these distributions on non-numeric domains. Even on numeric data, smooth distributions with closed mathematical descriptions are often poor representations of the actual data. Deviations from the equal frequency assumption and the argument independence assumption are discussed further in Sections 3.1.7 and 8.1.

3.1.4 Estimating set sizes

The estimated number of clauses in a set of resolvents is the product of the probability of successful unification with the number of attempted resolutions, which in turn is just the product of the estimated numbers of clauses for the parent sets. Thus if patterns P_1 and P_2 resolve giving P_3, and the probability of the first literals of two random instances of P_1 and P_2 being unifiable is written $Prob(Unify(P_1, P_2))$, with $N(P)$ being the number attached to a pattern P, then we have

$$N(P_3) = N(P_1)N(P_2)Prob(Unify(P_1, P_2)) \quad (1)$$

Under the two assumptions mentioned so far, if $d(x)$ is the number of values in the

domain of x, the probability of successful unification is

$$Prob(Unify(P_1, P_2) = \prod_{x \in X} d(x)^{-1}$$

where X is the set of constants and bound variables in the first literal of P_1 that are to be unified with constants or bound variables of P_2, but excluding cases where both are constants (because then, if they are not equal, the simulator can tell directly that the unification will fail, and if they are equal, the probability of success will not be affected). If there be some free variable that occurs more than once in the arguments of P_2, then all but one of the constants or bound variables of P_1 that unify with it must also be in X, since successful unification will require that all of them unify. For example, if P_1 is $A(U, V)$ and P_2 is $A(x, x)$ then V must unify with U, and so one or other of them must be put in X.

A better way to look at this is to consider the sets of bound variables in P_1, P_2, and P_3. Denote these by $BV(P_i)$. The set of bound variables that should be used to calculate the unification probability is

$$X = (BV(P_1) \cup BV(P_2)) \setminus BV(P_3)$$

Any bound variable that appears in P_1 but not in P_3 must have been unified with a constant or bound variable of P_2 (or perhaps of P_1), and so will contribute to the probability of the unification failing. One which does appear in P_3 either was unified with a free variable, which always succeeds, or else with another bound variable from P_2 or P_1, which therefore does not appear in P_3, and so the contribution of this unification to the probability can be assigned to the other variable. So each bound variable that appears in one of P_1 or P_2 but not in P_3 contributes exactly once to reducing the probability of successful unification. We assume, of course, that the variables of P_1 and P_2 have been standardised apart.

But this analysis only applies if all the bound variables of the first literal of a pattern also appear in one of its other literals; if some do not, they may disappear from the resolvent unnecessarily. In the example used earlier, when $B(R, S)$ resolved with $\neg B(V, z)C(U, z)$, there was such a bound variable, namely V; had P_2 been $\neg B(V, z)C(U, V, z)$ then P_3 would

have been $C(U,V,S)$ instead of $C(U,S)$, though the probability of successful unification would be the same. So it is not correct to write

$$Prob(Unify(P_1,P_2)) = \prod_{x \in BV(P_1) \cup BV(P_2) \setminus BV(P_3)} d(x)^{-1}$$

However, if we define P_3' to be the pattern P_3 augmented with the literal obtained by applying the unifying substitution to the first literal of P_1 or P_2, we can say

$$Prob(Unify(P_1,P_2)) = \prod_{x \in BV(P_1) \cup BV(P_2) \setminus BV(P_3')} d(x)^{-1} \quad (2)$$

In this example, P_3' would be $\neg B(R,S)C(U,S)$, so that $BV(P_3')$ would be $\{R,S,U\}$. As before, $BV(P_1) \cup BV(P_2)$ would be $\{R,S,U,V\}$, so we would get

$$Prob(Unify(P_1,P_2)) = d(V)^{-1}$$

which is correct, and can be used in equation 1.

3.1.5 Duplicate answers

These disappearing bound variables cause another problem. While each pair of instances of P_1 and P_2 will always generate a new instance of P_3', it may happen that some of these instances differ only in the binding of a bound variable that appears in P_3' but not in P_3. Since the simulated resolution generates P_3 and not P_3', duplicates of some instances of P_3 can occur. This would happen in our previous example with facts like $A(2,1)A(2,3)B(1,4)B(3,4)$, which would in fact cause $C(2,4)$ to be generated twice: once with $V = 1$ and once with $V = 3$. If instances of P_3 are to be stored in the database, these duplicates will presumably be detected and eliminated, reducing the number of clauses like P_3 that are available to subsequent resolutions. This reduction has been analysed by Smith [Smi85] as follows:

Let m be the number of resolvents generated before elimination of duplicates, namely $m = N(P_3')$ in our notation. Let

$$g = \prod_{x \in BV(P_3)} d(x) \quad (3)$$

be the number of potentially distinct instances of P_3 and let

$$h = \prod_{x \in BV(P_3') \setminus BV(P_3)} d(x). \tag{4}$$

so that gh is the number of potentially distinct instances of P_3'. Then the probability of a particular instance of P_3 being generated at least once, and hence stored in the database, is

$$Prob(UniqueInstance(P_3', P_3)) = 1 - \frac{\binom{gh-h}{m}}{\binom{gh}{m}}. \tag{5}$$

We now obtain

$$N(P_3) = g \times Prob(UniqueInstance(P_3', P_3)) \tag{6}$$

replacing equation 1.

Strictly speaking, this analysis applies only when m is an integer and is known accurately rather than estimated. We will often obtain values for m that are not integers, because they are the expected value obtained from some probability distribution for m. While it is easy to re-express the binomial coefficients using Stirling's approximation, so that we do not embarrass ourselves trying to take the factorial of a non-integer value for m, this does not avoid a more fundamental difficulty. The quotient of the binomial coefficients is not a linear function of m, and so its expected value will in general not be equal to its value computed at the expected value of m. This problem is complicated by the interdependencies between gh and m; see Section 8.1 for further discussion.

In considering resolvents that are stored, we must also take account of the possibility of duplication across different rules, as well as from different inputs to the same rule. This will lead to different pairs of patterns giving rise to resolvent patterns that are the same up to renaming of bound variables. We shall assume (the *rule independence assumption*) that the probability of a particular resolvent being generated from a given pair of parents

is independent of whether it was generated in any other way, so that the results of different deductive paths interfere purely at random. Thus, if the quantity g is defined as above, and if $g/2$ instances of $C(U,S)$ are produced by resolving $B(R,S)$ against $\neg B(V,z)C(U,z)$, but there is also a resolution of $D(P,Q)$ against $\neg D(t,W)C(W,t)$ that is estimated to produce $2g/3$ such instances, then the simulator will conclude that $g/3$ of them are produced twice, for a net total of $g/2 + 2g/3 - g/3 = 5g/6$.

So for patterns of clauses that will be stored in the database, we now obtain

$$N(P_3) = g\left(1 - \prod_{P_1,P_2}(1 - Prob(UniqueInstance(P_3', P_3)))\right) \tag{7}$$

where the second product is taken over all pairs (P_1, P_2) that resolve giving P_3, and P_3' is obtained as described above. We should point out that for each different such pair there may be a different P_3'. For patterns of clauses that will appear only on the agenda, so that duplicate instances are left alone, we have

$$N(P_3) = \sum_{P_1,P_2} N(P_1)N(P_2)Prob(Unify(P_1, P_2)) \tag{8}$$

In either of these cases, it is usually wise to treat the union of sets having identical patterns (up to renaming of free and bound variables) as a single set for the purpose of simulating further resolutions.

Given the set of clauses in R, and the patterns and estimated sizes for sets of terms in F and G, we can now iteratively obtain descriptions of all the sets of resolvents generated. Coupling this with knowledge of the cost of generating an individual resolvent, we will have an estimate of the cost of the strategy. This iterative approach is clearly not adequate for dealing with recursive rules in R; it will loop on backwards inference in the same way as a LLNR resolution system would do at run-time. In general, it is not possible to estimate the cost of every recursive computation, because this would imply a solution to the Halting Problem. However, there may exist problem classes for which useful estimates can be made.

3.1.6 Single answers

All of the above equations apply only when backwards inference is expected to return all the answers to the goal. If this is not so, for example if only the first answer returned is wanted, then the implicit assumption that all resolution operations that are syntactically possible will be done becomes false. To handle this case, the estimates would need to be adjusted by the probability that an answer had already been obtained by the time the resolution was to be attempted. This probability does not bear any simple relationship to the expected number of answers returned, which could reflect a small chance of many answers or a larger chance of few answers. So the estimation would become very much harder in the single-answer case, and rule ordering would have to be taken into account.

3.1.7 Other useful information

Unequal frequencies

If the distribution of actual constants in the facts and goals does not conform to the equal frequency assumption, the estimated numbers of resolvents may be almost arbitrarily badly wrong. Two safety mechanisms are possible for this. The first is to specify that some value is going to be over- or under-represented; this could be done for several values if necessary. The second is to allow the user to give the probability of successful unification directly.

The current implementation does allow for probability distributions in which individual values have higher or lower than average probability; this is handled by having a pattern P' which is a substitution instance of another pattern P, with the distinguished value (a constant) substituted for the relevant bound variable. We can then compare the number of clauses expected to match P' with the number obtained from the number attached to P via the equal frequency assumption, and attach the difference of these numbers to the pattern P' to represent a "maverick" set of clauses. With some additional effort one could implement a mechanism for indicating that some subset of a domain is to be uniformly over-

or under-represented. Piatetsky-Shapiro [PC84] suggests a way of doing this for domains that allow of a total order on their values, and gives some bounds on the accuracy achievable.

However, this extra information can slow down the simulator severely, and it may be better to allow the user to tell the simulator that some pairs of bound variables are related so that the probability of a successful unification involving them is higher or lower than the equal frequency assumption would indicate. For example, in a Computer Science Department where all the students have ages between 12 and 50, the probability of a random student being the same age (in years) as another may actually be 0.2 or so. The simulator can simply use this value instead of subtracting 12 from 50 and taking the reciprocal.

Argument dependencies

It is likely that equality between two values will in fact depend on the values of other variables. For example, the probability of two students being the same age tends to increase if both of them are known to be undergraduates, and may well have a different value (but still above average) if both are graduate students. Ideally, the probability of unification as given by equation 2 should be modified to take account of this.

Two kinds of obstacles arise now. First, the aim of this whole simulation, and the optimiser of which it forms a part, is to make logic programming easier and more efficient, and this will not be served if the author or user of a logic program is pestered with many questions about whether some value is strongly or weakly dependent on some other value or set of values. It seems reasonable that functional dependencies, where one value determines another, should be made known to the simulator, but the most that seems advisable in connection with other dependencies is that the human should have an opportunity to volunteer information.

The second source of difficulties is that the simulator would be complicated and slowed if it had to remember what had been proved about a given term, such as whether a student-valued variable in some clause was known to represent an undergraduate. This becomes

acute when incoherent strategies are being considered. Suppose, for example, that there were two rules

$$A(x) \,\&\, B(x,y) \;\Rightarrow\; C(y)$$
$$D(x,y) \,\&\, E(x,y) \;\Rightarrow\; B(x,y)$$

If both of these were used backwards, then attempts to prove $D(x,y)$ would be made only with x values such that $A(x)$ had already been proved in the first rule; if the second rule were used forwards, nothing would be known about the x values. Thus, if some backwards rule had to be invoked to prove $D(x,y)$, the unification probabilities in the resolutions done under this rule could depend on the direction in which the second rule was used.

When only coherent strategies are being considered, this is not a problem, because if the rule that proves $D(x,y)$ is invoked backwards, then the other rule is its successor and so must also be used backwards. The dependence noted above is now of no importance. However, if the order of the input literals of a rule is allowed to vary, then by merely interchanging the input literals of $A(x)$ & $B(x,y) \Rightarrow C(y)$, we can cause the other rules to be invoked with different sets of values for x.

It does seem that, if optimisation is restricted to coherent strategies and the order of input literals is not subject to change, unification probabilities could be obtained by direct observation of a few sample runs of the logic program being optimised. Since these probabilities would be independent of the strategy being used, they could then be fed to the simulator to produce cost estimates that could safely be used to select the optimal strategy as discussed in Chapter 4. It would be necessary to observe each rule's behaviour both when used forwards and when used backwards, since even in the coherent case these can lead to different unification probabilities. Some of the strategies that would have to be used for these observations would be non-optimal, perhaps badly so, and the observation might become very expensive. Anyone who contemplated expending this much effort on optimisation would be well advised to consider re-ordering of input literals, and perhaps

incoherent strategies, as these can give much lower run-time costs. See also the discussion in Section 8.2 about adjusting estimates in the light of experience.

Number of answers known

As noted above, it is important to inform the simulator of functional dependencies between arguments, such as where $P(X, Y, z)$ is known to yield just one value of z for any pair of bindings of X and Y. It may be just as important, and will clearly be useful, to tell the simulator about predicates where the number of answers is not necessarily 1, but is still known in advance, the standard example of this being that a human has exactly two parents. If such information is available, the equations for $N(P_3)$ given above can be overridden. Even if only an upper bound on the number of answers is available (one or both parents may be dead), this is of use when upper bounds on costs are wanted.

Information about the number of answers is particularly important when a subgoal is to be solved by executing a procedure associated with its predicate symbol, rather than by database lookup or backwards inference. Procedures used for this purpose generally do not unify the subgoal against anything; the only statistic that it even makes sense to supply is the expected number of answers returned.

3.2 Execution Model and Cost Estimates

The cost of the resolution process can be broken down into the following components:

1. choosing a clause to resolve on
2. finding clauses that may resolve with it
3. performing the unifications
4. constructing the resolvents, when unification succeeded
5. storing the resolvents, if needed.

Step 1 is assumed to take negligible time, since we assume the clause to be drawn from the agenda.

Steps 2 and 3 are not necessarily distinct, since one way to accomplish their effect would be to try unifying the first literal of the chosen clause with that of every other clause in the database. We expect, however, that some index into the database would be used to pick a set of clauses likely to resolve with the chosen one. The set of "candidate" clauses found by indexing could then be put through a filtering step to detect clauses that would not in fact resolve with the chosen clause, but such a filter might as well be regarded as the first part of the unification algorithm, and so could be part of step 3 as easily as step 2. The cost of unification itself would then be the product of the size of the candidate set with the average cost of unifying the two literals, which has been shown to be linear [MM82].

Indexing must be done even for a lookup that returns no answers, so we must add the cost of indexing, which is independent of the number of answers, but probably proportional to some function of the size of the literal being indexed on. The upshot of all this is that the total cost of steps 2 and 3 can be expected to be a linear function of the number of clauses returned, and some function of the first literal of the clause taken from the agenda. We assume that the simulator knows enough about the indexing scheme to be able to predict the proportion of candidate clauses returned by step 2 to answers returned by step 3 for each possible literal; this may depend on the number of clauses stored that match each pattern, but these numbers can be computed, even for clauses generated during the deduction, using the equations in Section 3.1.

The cost of step 4 (generating resolvents) can easily be described in terms of the number of variables and the number and complexity of the literals involved. It is not hard to do the substitutions necessary for this in time proportional to the sum of the number of variables in the unifying substitution and the total number of appearances of variables in the parent clauses. However, for clauses that are not going to be stored

in the database but only kept on the agenda for a while, it is generally better to refrain from substituting all the variable bindings into the literals of the resolvent clause. An alternative representation for generated terms, using the original terms plus a vector of variable bindings, is described by Warren in [War77], where the cost of generating a new term is said to be no more than the number of distinct variables in it, and presumably also no more than the number of variables that received bindings in the preceding unification. Indeed, if the unification algorithm is allowed to set bindings for variables as a side-effect, the cost of generating the resolvent is just that of an **Append** operation (or a `Cdr`, if one clause was a unit) plus perhaps some "housekeeping" overhead.

It is admitted that the use of this representation can complicate subsequent unifications because of the need to follow chains of references from variables to their current values (i.e. bindings) before unifying them with anything. This cost should properly be added to the cost of indexing rather than that of unification, since the dereferencing has to be done independent of how many successful unifications take place. Warren suggests that, whenever a backwards rule is resolved against a clause on the agenda, the values found for the variables in the first literal of the agenda clause should be kept locally in a stack frame, which disappears when the last literal from the rule has been resolved away.

So the chains of references will always go via bindings created during the execution of this backwards rule to a cell in the current top frame of the stack, where they will find either a constant value or a single further reference to some variable (whose value can mow be modified as a side-effect of unifications done while processing this rule). This allows the chains to be kept short, and removes any dependence of the costs of unification and resolvent generation on the contents of lower stack frames, thus easing the simulator's task considerably as we shall see later (Section 3.4). Of course, the cost of generating a resolvent whose database parent is a backwards rule is elevated, but this is not hard to simulate. It is also easy to simulate the construction of chains of references and to check their lengths

when simulating the dereferencing operation.

The cost of step 5 (storing resolvents) will include the cost of dereferencing variables, i.e. finding their bindings, since we assume that clauses stored in the database are fully substituted. This is acceptable if all such clauses are short; if not, another representation might have to be considered. This part of the cost will also depend on the algorithm used to index the database. We expect the cost per resolvent to be proportional to some function of the length of the resolvents. This function will presumably have roughly the same value over all resolvents matching a given pattern, so the total cost of steps 2, 3, 4, and 5 for resolutions corresponding to some pair of patterns will be a linear function of the number of resolvents generated.

In the NP-completeness proofs below, where comparisons between costs become important, we shall use symbolic expressions rather than commit ourselves to any particular assumptions about the costs of various operations. We will, however, assume that the indexing is done the same way for all predicates, so that only the length of a literal (or clause) affects the cost of indexing on it and the proportion of the candidates returned by step 2 that were succesfully unified in step 3. In a real implementation of this work, the cost estimator could be equipped with knowledge about the indexing scheme and other details, which could change certain elements of the cost relative to others by a constant factor. The NP-completeness proofs are impervious to such changes.

When an attached procedure is used instead of resolution to solve some subgoal, conventional techniques can be used to estimate its running time.

3.3 Breaking down the costs

We shall see in Chapter 4 how the choice of an optimal strategy becomes easy when each rule has well-defined costs for using it forwards and backwards, which are independent of anything done with the other rules. It turns out that this can be done only in the

case where input literal order is fixed, strategies are required to be coherent, and no rules generate duplicate answers; we shall see later that this is also the only case examined in this dissertation which does not yield an NP-complete problem. Intuitively, we may say that when the cost of one component of a strategy depends on the use of other components, it becomes hard or impossible to decide that some strategy is more expensive than another without looking at them both in detail. But when component costs are fixed, we can prove statements of the form "Any strategy using component X costs more than one using Y instead of X", which allow drastic pruning of the search space.

Since a clause pattern can be derived via a sequence of resolutions involving several rules from R, we need some way of assigning the costs associated with a set of clauses to one or more rules and/or goals, so that the total cost of a strategy is equal to the sum of costs over all the rules (plus, perhaps, costs attached to the goals in G), and so that the cost numbers for each direction of each rule accurately reflect the consequences of using that rule in that direction.

The key to assigning costs is to consider the first literal of a derived clause or pattern. Facts have only one literal, so the literals of a derived clause are all descended (via some number of substitutions) from a literal of some rule or goal. Each derived clause will be associated with the rule or goal from which its first literal came. For the empty clause, this breaks down. The current implementation finesses this by assuming that there are no costs associated with an empty clause; in general there will be some cost associated with returning a set of variable bindings, constituting an answer to some goal, and this cost could be assigned to that goal. However, since the same answers must (should) be returned by every strategy, the cost of returning them does not vary across strategies, cannot be optimised over, and so is not important.

For non-empty clauses, the cost of generating them and storing them (if they are stored) is assigned to the rule or goal with which they are associated. Regarding the cost of indexing and unification (steps 2 and 3 above), the treatment of these varies slightly depending on

whether forwards inference is done by putting facts or rules on the agenda. If the latter, then all costs for steps 2 and 3 of a resolution are assigned to the parent clause from the agenda, or to its associated rule or goal as necessary. Otherwise, an exception is made when a fact (unit clause consisting of one output literal) is taken off the agenda: the cost of step 2 (indexing) is assigned to the rule, if any, which generated the fact, while the cost of step 3 (unification) is split up among the rules or goals whose first literals unify with the fact. For facts which are not generated by any rule, namely facts in F, the assignment of indexing cost is unimportant, because this cost is incurred equally by all strategies.

3.3.1 Costs of forward inference

Assuming coherence and forward inference, the cost of a rule is simple to calculate. Consider the clause $\neg A(x,y)\neg B(y,z)C(x,z)$ as before, with fact patterns $A(U,V)$ and $B(R,S)$. Coherence implies that the facts matching these patterns will already be available in the database.

We can use the equations from Section 3.1 to estimate the cost of resolving the rule against facts like $A(U,V)$. For illustrative purposes only, we shall set the cost of each of the elementary operations discussed above to 1 unit. If there are 5 of these facts, then each of them will resolve against the rule, giving 5 resolvents like $\neg B(V,z)C(U,z)$, at a cost of 1 unit for indexing, 5 units for looking up the facts, and 5 for generating the resolvents, or 11 units total. We likewise estimate the cost of resolving one clause like $\neg B(V,z)C(U,z)$ against facts like $B(R,S)$: if there are 8 such facts, and R ranges over a domain of 10 values, then we can estimate the number of successful unifications to be 0.8, with a total cost of $1+0.8+0.8 = 2.6$ units. Since there will be 5 such clauses, this gives 13.0 as the second part of the cost of the rule, and 4 as the number of facts like $C(U,S)$ deduced. Adding the cost of storing these into the database, namely 4 units, we get a grand total of $11 + 13 + 4 = 28$ units. Admittedly some of these four facts may be duplicates of each other, but since the code for storing them into the database must be invoked just to discover this, the cost is

still 4 units.

To illustrate the effect of changes to the order of input literals, assume now that they are exchanged in this rule, giving $\neg B(y,z)\neg A(x,y)C(x,z)$. We then get 8 resolutions of facts like $B(R,S)$ against the rule, generating 8 resolvents like $\neg A(x,R)C(x,S)$ at a cost of 17 units. Each of these has a probability 0.1 of successfully resolving against one of the 5 facts like $A(U,V)$, and so will give on the average 0.5 resolvents like $C(U,S)$ at a cost of $1 + 0.5 + 0.5 = 2.0$ units. Thus the total cost comes to $17 + 8 \times 2 + 4 = 37$ units (there are still, of course, the same four answers). Not only is this larger than the cost with the original ordering, but most of the subtotals which make it up have changed.

In the incoherent case, it may be that the inputs for this rule are to be found by backward inference, so that the cost of looking them up in the database must be replaced by the cost of looking up the rule(s) that deduce them. Moreover, the answers returned may include duplicates, causing increases in the numbers of resolvents that contribute to the cost of this rule. Thus the cost of using the rule forward will depend on whether its predecessors are used backward, and if so, whether they generate duplicates.

3.3.2 Costs of backward inference

To estimate the cost of using the clause $C(x,z)\neg A(x,y)\neg B(y,z)$ backwards when it is triggered by some goal or subgoal clause whose first literal unifies with $C(x,z)$, we can proceed similarly, namely assume that all the facts like $A(U,V)$ and $B(R,S)$ are in the database, and estimate the number of intermediate clauses and resolution steps. Note that the clauses generated after all the input literals of this rule have been resolved away will not "belong" to this rule, but to the rule or goal which invoked it, and so the cost of generating them will not be charged to this rule. Several difficulties now arise, though most of them are easily handled if the strategy is required to be coherent and the order of input literals is fixed.

First, the invoking clause may well have been longer than one literal, so that the first

resolvent generated will begin with literals descended from $\neg A(x, y) \neg B(y, z)$ but will have other literals after them. This longer resolvent may be more costly than the ones generated during forward inference; if so, the cost of using the rule will depend on how it was invoked, i.e. what kind of clause unified with its output literal. This in turn depends on whether the invoking clause was generated by backward chaining all the way from a goal pattern, or from some rule that was being used forwards.

In the coherent case, none of the rule's successors can be used forwards, so the invoking clause must have come from a goal in G, and we can be sure in advance what the resolvents will look like and how much each of them will cost to generate. In the incoherent case, different strategies will give different invoking clauses, so we must adjust the cost estimate for each new strategy, or else put up with estimates that may be inaccurate because the lengths of resolvents are not known precisely. This will be considered further in Chapter 6. If literal order can be changed, then clearly the length of an invoking clause beginning with a given literal is not fixed, because other input literals of the same rule might come before or after it. This makes it impossible to estimate costs accurately even in the coherent case.

Secondly, as remarked above, the cost (incurred by other rules) of looking in the database for unit clauses matching $C(x, z)$ can be deleted if the simulator knows that such clauses are not going to be there. Even though this lookup cost will have been charged against rules which are successors of the rule we are considering, the saving is directly attributable to the use of this rule in the backwards direction, and can be credited to the decision to use it so. The number of such lookups the successor rules have to do will depend on whether they are used forwards or backwards, and so the total saving would vary across incoherent strategies. Coherence requires that they all be used backwards if the original rule is, so this saving is the same for all coherent strategies provided that literal order is fixed. But if it is not, the number of facts looked up can change drastically.

Thirdly, since answers for $C(x, z)$ are no longer assumed to be stored, so that duplicates cannot be eliminated, there may be an increase in the number of answers obtained

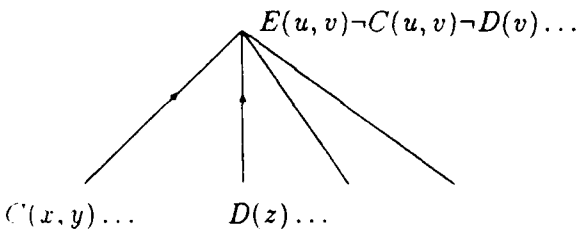


Figure 3: Rules affected by duplicates

if this rule is changed to backwards, leading to higher costs for other rules. Rules such as $E(u,v)\neg C(u,v)\neg D(v)$, namely successors of the current rule, will be affected regardless of their directions, unless C appears only in their last input literal. Predecessors of this successor whose output literals unify with an input literal occurring after the $C(u,v)$ literal, such as the $\neg D(v)$ in this example, will also be affected, since there will be more resolvents beginning with a $\neg D$ literal to unify with these rules. See Figure 3. Even in a coherent strategy, the directions of these rules are not determined by the direction of the original rule. Thus the extra cost incurred by other rules due to duplicates generated by this rule is not even the same across all coherent strategies. Incoherence and re-ordering of input literals will make this worse.

Finally, the total cost of using the rule for backward inference will depend on the number of clauses that invoke it. If all these are generated entirely by rules used in the backwards direction, then their number is determined by the structure of the set R of rules and the set G of goals. But if some successor of this rule is to be used forwards, the number of clauses it generates will depend on F rather than G. Thus, the number of invocations of a rule is a function of the directions in which other rules are used, namely its successors and their successors and so on. In the coherent case with fixed input literal order, these directions are fixed by the assumption that this rule is used backwards, so this part of the cost calculation

is the same for all such coherent strategies. Again, changes to input literal orders, which change the structure of R, can radically alter the numbers of clauses, and indeed the sets of variables that are bound when a particular literal is being solved.

3.4 Cost of Obtaining Estimates

In the cases where fast algorithms exist for finding the optimal strategy given a set of cost estimates, it is reasonable to expect that the total time spent in optimisation will be dominated by the time spent estimating the costs, and this is borne out by experience. Where a search has to be done through many partial strategies, each one must be evaluated, i.e. some form of cost estimation must be done there, and this is likely to take much longer than generating it. So it is very important that cost estimation should not take an unduly long time.

3.4.1 Cost of clause sets

Adding up numbers as described in Section 3.3 is comparatively easy; the cost of the simulation will clearly be dominated by the cost of generating the clause patterns and computing the sizes of the sets. Decompose this.cost further into the product of the number of clause sets and the cost of each one.

The cost of estimating the cost of an indexing operation is, presumably, linear in the size of the literal being indexed on, since it just involves looking at each argument position and checking whether there is an index on it. The cost of using the index can be assigned constant cost, since it will depend on the size of the index, which can be computed once and for all while the numbers of clauses stored in the database are being computed.

Simulating a lookup involves doing indexing and unification on the database of clause patterns. The cost of this indexing should be no more than linear in the size of the first literal of the pattern taken off the agenda, and will also have some dependence on the

number of clause patterns, probably logarithmic. The unification cost depends only on the size of the first literal, and good unification algorithms are known so that this cost will not be very large; the additional processing needed to keep track of bound variables will be at most linear in the number of arguments in the literal.

The simulator must also figure out how many clauses will be returned by unification. The cost of doing this will be, for each clause pattern returned by the simulated unification, just the cost of figuring out the unification probability in each case. This cost is proportional to the arity of the predicate involved. So here the main influence on the cost is the number of patterns of clauses to be stored in the database.

To simulate the generation of clauses we must generate a new clause pattern, probably by applying to the old patterns the substitution obtained from simulated unification. The time this takes will be proportional to the sum of the lengths of all the literals in the new pattern. This may be quite long, but as long as there are no recursive rules, it cannot be longer than the sum of the lengths of all literals in R and in the longest clause in G, and probably less than this. It would not take long to find the longest possible clause that could be generated.

3.4.2 Number of clause sets

In simulating a rule used forwards, the number of clause sets generated will be equal to the number of literals in the rule, unless there are several clause sets whose patterns unify with the same input literal of the rule. Since facts have no free variables, the only way this can happen is for there to be fact patterns with constants in them as well as bound variables; several of these, differing only in their constants (or in the presence of a constant where another pattern had a bound variable) could all unify with the same input literal. So we see that, as asserted in Section 3.1.7, mentioning constants in the description of F can slow down the simulation, by causing more clause sets to appear. If a rule has k variables, and there are n_i constants that could be unified with the ith variable, then up to

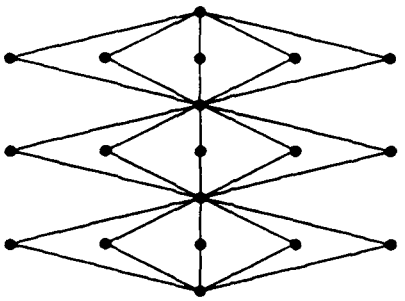


Figure 4: Reconvergent fanout

$\prod_{i=1,k} n_i$ different clause patterns could be generated where there was one before, so that the time taken by the simulation could be exponential in the size of the input given to it. Simulating something in great detail is often worse than executing it directly. Note also that the product $\prod_{i=1,k} n_i$ can increase greatly in the presence of long rules, or predicates of high arity. Ugly programs really are harder to optimise!

Simulating a rule used backwards is worse than the forwards case: for each pattern that invokes the rule, we may do as much work as we had to do to simulate using the rule forwards, or even more. Since a subgoal clause pattern can have many more variables in it than a pattern generated from a single rule used fowards, the potential number of patterns that are distinct on account of constants appearing in them becomes larger. This effect is mitigated slightly by the fact that some of the variables in a subgoal inherit their bindings from the next higher goal, and so the number of different constant values that could appear is not quite as big as the number of appearances of variables. But the number of invoking patterns is large: for every path through the rule graph from the rule to some goal, there is a potential clause generated by invoking the rules along that path, which the simulator must consider.

So even in the complete absence of constants in the description of F or G, the number of distinct goal patterns can be as large as the number of paths through the rule graph. If the graph has the property known to circuit designers as *reconvergent fanout*, namely there

are links from node A to many nodes B_i and each B_i has a link to C, then the number of paths through the rule graph can be much larger than the number of rules. For if this construction is repeated with C, D_i, and E, then for each pair of nodes (B_i, D_j) there is a distinct path from A to E: if we repeat this m times with n paths each time, we can get a graph with n^m paths through it but having only $mn + m + 1$ nodes. See Figure 4, where the case $m = 3, n = 5$ is shown. Logic programs frequently do display reconvergent fanout, though usually not to such an extravagant extent.

To summarise, the main threats of exploding costs for estimating the cost of a strategy come from the following sources

- many constants appearing explicitly in descriptions of F, R, and G
- many patterns for clauses to be stored in the database
- long clause patterns (caused either by long rules or long paths through the rule graph)
- many paths through the rule graph

The first of these is entirely the user's fault; the second, when not caused by the first, stems from large numbers of rules or of fact sets, and so is also the user's fault, in the sense that it is not reasonable to expect to simulate such a logic program both accurately and cheaply. The length and number of clause patterns generated are only indirectly the user's responsibility, and something can be done to mitigate their adverse effects.

3.4.3 Collapsing clause sets

The effect of reconvergent fanout will be that the simulator will generate many sets of clause patterns whose first literals are identical up to variable renaming, even though the later literals may differ. When two such patterns are resolved against the same backwards rule, if this rule has n input literals, we will get clause patterns whose first n literals are the same up to variable renaming, while the rest may differ. Until all of these literals have

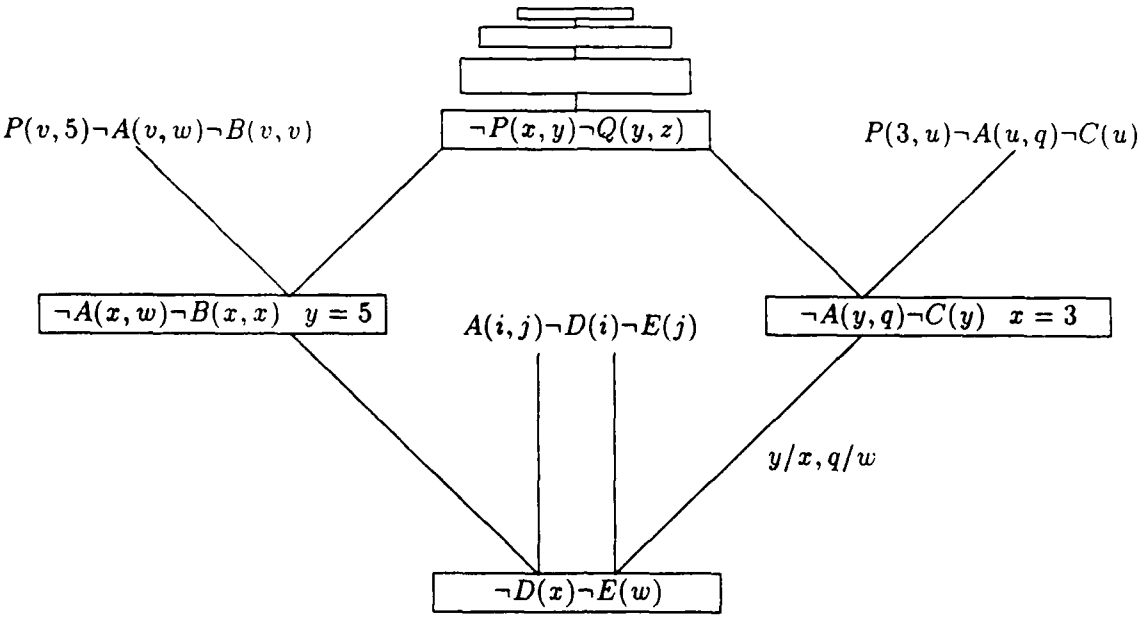


Figure 5: How tails share structure

been resolved away, every resolution that can be done with one of these patterns can also be done with the other. Their simulated costs will also be the same, except for possible differences resulting from different lengths of the patterns. We can take advantage of this by forming equivalence classes of patterns whose first few literals are alike and letting the simulator work mostly with these classes rather than the patterns themselves.

To form the classes, chop up the patterns into a "head" and a "tail", where the head consists of those literals that are descended from input literals of the same clause (rule or goal) as the pattern's first literal came from, and the tail is all the other literals. The tail will in fact be represented as a clause pattern in its own right, with a head and a tail, and so the data structure is recursive. A goal clause is the base case, being created with no tail. The results proved in Section 2.2.1 show that the head of a resolvent always consists of the remaining literals of one parent and the tail is the remaining literals of the other parent (all suitably substituted).

Whenever a new clause pattern is created, the simulator checks to see if any other exists with a similar head, and if so, adds the new pattern's tail to the list of tails behind the old head. An equivalence class of patterns is thus represented by a single head and a list of tails; each tail on the list (except the oldest one) is labelled with a substitution which, when applied to the head, gives back the head that was originally attached to this tail. Figure 5 shows how the clause structures created this way mimic the reconvergent fanout found in the rule graph. In this figure, the cclause heads are shown boxed, and the clauses outside the boxes are the rules. When the last literal of a head is resolved away, simulated resolution must return the tails that were attached to that head (suitably substituted) rather than a pattern with an empty head.

In the absence of explicit constants in the descriptions of F and G, the maximum number of heads for each rule is no more than the number of literals in the rule times 2^v, where v is the number of variables in the output literal of the rule. The factor of 2^v comes in because each such variable can be either free or bound when the rule is invoked. In practice we expect that most logic programs will not display the full range of possibilities in this respect, until arbitrary orderings of the input literals are considered. Even then, as long as the arities of predicates are not very large (as they usually are not), the total number of heads that arise can be far smaller than the number of paths through the rule graph. The presence of explicit constants will multiply the number of distinct heads by the quantities discussed above, but since the heads are short compared with full subgoal patterns, the worst of these do not arise.

When the simulator estimates the costs of operations on clauses represented as a head and a list of tails, it needs to know the costs of generating resolvents, which can depend on the lengths of these clauses if full substitution is done. See the discussion in Section 3.2. These lengths can be different for two clause patterns with the same head but different tails. If each head carried a list of the lengths of the tails attached to it, we would once again be distinguishing individual patterns, which we do not want to do, since the number of them

can be large.

One solution to this is to let each head carry the sum or some other aggregate function of the lengths. If the dependence of the cost on the length is linear, then we can simply take the sum (weighted by set size) of the lengths of the tails, and the result of using this to estimate the costs will be accurate. When the simulator generates a new head, the total length of its tails is the average length of the patterns in the equivalence class of the agenda parent of the new resolvent. This average is given by the expression $h - 1 + t/s$ where h is the number of literals in the head of this parent class, t is *its* total tail length, and s its total set size. When a new clause is joined to an existing equivalence class, we simply add the total length of its tails to that for the class.

There is another solution, which is likely to be more expensive. The maximum possible length of any tail is no more than the total length of all the rules and goals in R and G, unless there are recursive rules. So we could equip each head with a list of the lengths of all tails attached to that head, each length appearing only once and being paired with the total of the set sizes of all patterns belonging to this head whose tails have that length. The length of such a list would be bounded by the total length of R and G, which would be large but not exponentially large, and the time needed to estimate the length-dependent parts of the costs would be proportional to the length of the list.

Chapter 4

The Optimal Coherent Strategy

In this section we examine the case where a rule that is used forwards requires all its predecessors to be so used. If there are no rules generating duplicate answers (the same conclusion being drawn from different sets of inputs to the rule), then the optimal set R_f of rules to be used forwards can be found by any linear programming method. We also mention a specialised method that is very fast and sacrifices little generality.

If some rules can generate duplicates, the problem becomes NP-complete. By treating such rules separately from others, we can find the optimal set in time bounded by a polynomial in the total number of rules times an exponential in the number of "bad" rules (and much more quickly than this in most cases).

4.1 No Duplicate Answers

The number of coherent strategies on a set R of rules can in general be exponential in the number of rules in R. To see this, simply consider a set of n rules none of which is the predecessor of any other. Any subset of these rules would constitute the set R_f of rules used forwards in a coherent strategy, and there are 2^n such subsets. We shall show that it is often possible to find the optimal coherent strategy in time very much shorter than would

be required to examine all possible strategies.

It was shown in Chapter 3 that under the two restrictions just mentioned (coherence of strategy and no duplicate answers), the cost estimates $e_f(x)$ and $e_b(x)$ have only F, R, and G as implicit arguments. In particular, they do not depend on the directions of other rules. Then the problem of finding the set of nodes to be used forwards in a least total cost strategy is just an integer programming problem. Namely, recalling the definition of $v(x)$ from Section 2.4, the total cost of a strategy (represented as a set of values for the $v(x)$) is

$$\sum_x v(x)e_f(x) + (1 - v(x))e_b(x)$$

and the constraint on rule directions turns into the inequality

$$v(y) \geq v(x)$$

for each predecessor y of x. If this expression for the cost is minimised subject to these inequalities, the resulting values of the variables $v(x)$ give the optimal strategy.

4.1.1 The linear program

This integer program is in fact a linear program, and hence solvable in time polynomial in the number of constraints, which is polynomial in the total number of literals in R. General integer programming is NP-complete, and so is not believed to be solvable in polynomial time. To see that we really have a linear program here, it is necessary to prove that if the above constraints were augmented by the inequalities $0 \leq v(x) \leq 1$ for all x and the whole system solved as a linear program, the solution obtained would in fact give integer values to all the $v(x)$.

Suppose that the linear program can find an optimum with fractional values for some of the $v(x)$. Let k_1 be the lowest non-zero value taken by any of the $v(x)$, and k_2 the second lowest; let V be the set of variables $v(x)$ which take the value k_1. Then once the optimum has been reached, for any $v(x)$ in V, the constraints between $v(x)$ and any variable not

in V will all have the form $0 \leq v(x)$ or $v(x) \leq k$ for some $k \geq k_2$. It is clear that if k_1 and all the variables in V were simultaneously changed to k_2, or to 0, leaving the other variables unchanged, no constraint would be violated: the resulting set of values for the $v(x)$ would still satisfy all the constraints. But these changes correspond to moving in opposite directions in the solution space, and since the objective function is linear, the changes in it due to moving in opposite directions must be opposite. Since the original set of values for the $v(x)$ was assumed to yield the optimum, it is impossible for either of these changes to be an increase, so it must be that they have no effect on the objective function. So we could have obtained the same value for the objective by letting all the variables in V be 0. Iterating this construction, we can eliminate all non-integer values for the $v(x)$ without disturbing optimality.

Using a linear programming method such as the Simplex Algorithm [Dan65], which always obtains a solution at a vertex of the simplex determined by the constraints, we are assured of getting a solution to the integer program. The simplex algorithm does not have a polynomial time upper bound. We shall see later that another algorithm is available which does; in general, though, we could use any linear programming algorithm, and if a solution on a face or edge of the simplex is obtained, any vertex of this face or edge will yield an optimal solution.

4.1.2 Storage costs

Roussopoulos [Rou82a] has considered another strategy choice problem, different from this one mainly in the imposition of an upper bound on the total amount of storage available for the facts deduced (it is hard to tell whether he requires his strategies to be coherent). Such a bound can be added to this problem very simply, for if $s(x)$ is the space needed to store the results of x, and A is the total space available, we simply add the inequality

$$\sum_x s(x)v(x) \leq A$$

to the linear program. The same would apply to a time bound or to a bound on any expression linear in the $v(x)$, when combined with any cost function linear in the $v(x)$. Unfortunately, this causes the proof that the linear program can always return a solution in integers to break down, since changing several of the $v(x)$ from a fractional value to 1 might violate the storage bound. So the problem of finding the optimal coherent strategy subject to a bound on some element of the total cost appears to be harder than linear programming, and may be NP-complete.

Roussopoulos also expects that (translating into the terminology used in this paper) the only facts that will be stored permanently are those which are inputs to some backward rule. This means that facts which were deduced forwards only to be used as inputs to further forwards deductions would be erased from the database afterwards. Until now we have lumped space costs with time costs, thus assuming that every fact is stored. If there is no storage bound, the linear program can be modified to consider storage costs only for facts needed by a backward rule. Simply re-define the total cost of a strategy as

$$\sum_x v(x)e_f(x) + (1 - v(x))e_b(x) + v'(x)e_s(x)$$

where $e_s(x)$ is the estimated amount of space taken up by the facts deduced by rule x. The new variable $v'(x)$ is made to be 1 if x is a forward rule with a backward rule among its successors, and 0 otherwise, by the constraints

$$v'(x) \geq v(x) - v(y)$$

for all successors y of x. If x has no successors then $v'(x)$ is made identical to $v(x)$. The rule graph must also be extended to include nodes for the facts in F, each of which has zero values for $e_f(x)$ and $e_b(x)$ and 1 for $v(x)$. These changes roughly double the numbers of variables and constraints in the linear programming problem. The proof of all-integer solutions continues to hold, since the $v(x)$ can take exactly the same sets of values as they could in the original formulation of the problem, and each $v'(x)$ will take the lowest value it can, which will be 0 or 1.

Strictly, this trick, and indeed any version of the linear program that purports to take account of storage space, is only valid under the constraint that rules whose sets of answers overlap are always used in the same direction as each other. Without this constraint, the amount of extra space taken up to store the facts deduced forwards by some rule could vary depending on whether another rule had been used used forwards and had already deduced some of those facts. Duplication of answers across rules can have the same effect on the complexity of the optimisation problem as duplication of answers within a single rule, as discussed in Section 4.2.

4.1.3 The network flow model

The simplex algorithm lends itself easily to modification for the case where variables have upper bounds on their values, and has been coded and tried with good results. Its average case behaviour has been shown by Smale [Sma83] to involve a number of pivot steps that is, for a fixed number of constraints, proportional to the number of variables, although it is not clear how the proportionality constant is related to the number of constraints. However, it has a theoretically exponential-time worst case, and experience with the pilot implementation suggests that it might prove clumsy on large rule graphs. Papadimitriou [Pap85] has suggested a reduction of this problem to a network flow problem, a special case of linear programming which is known to be solvable in time proportional to the cube of the number of nodes in the network. Indeed, for sparse networks an even faster algorithm has been devised by Sleator and Tarjan [Sle80] with asymptotic running time $O(NA \log N)$ where N is the number of nodes and A the number of arcs.

The network for the case where storage costs are ignored is generated by adding to the rule graph two "dummy" nodes S and T, with arcs from S to every real node x and from x to T. The arc from S to x has capacity $e_f(x)$ and that from x to T has capacity $e_b(x)$. The original arcs from a rule to its successors have infinite capacity. In Figure 6, these arcs are

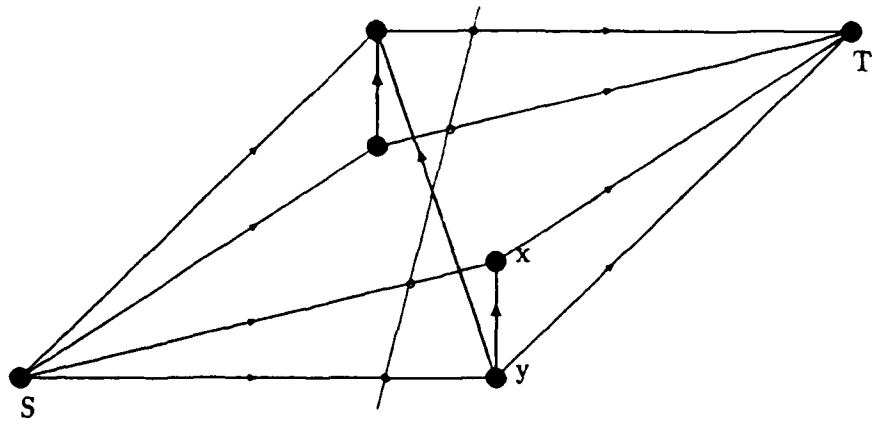


Figure 6: A network with four rules

represented by thicker lines. The choice of $v(x)$ is now reduced to the problem of finding the maximum flow through this network from S to T, as follows.

Consider a cut in this network, i.e. a set of arcs whose removal makes it impossible to find a path from S to T. Such a cut is shown in Figure 6, with small circles indicating where an arc is cut. Require the cut to have finite total capacity, so that it cannot include any of the arcs between nodes of the original graph. Thus if some node x is still connected to T after the arcs in the cut have been removed, and some other node y is a predecessor of x, then y is certainly also connected to T, via x. So the set of nodes that are still connected to T after a cut is closed under taking predecessors, and can be regarded as the set R_f in the strategy represented by this cut. The total capacity of the cut is clearly the sum of $e_f(x)$ over all nodes x whose arc from S has been cut plus the sum of $e_b(x)$ over all other nodes, and so it corresponds to the cost of the strategy. So far we have taken no precautions against a node being isolated from both S and T, which would correspond to an illegal strategy using a rule both forwards and backwards. But if we require the cut to have minimal capacity then each possible path from S to T will be cut exactly once, so each node will be connected to either S or T.

Now, since the maximum flow through a network is equal to the minimum cut required to disconnect the source and sink nodes, applying a standard network-flow algorithm will give us such a cut, corresponding to the least cost strategy. This approach has also been implemented, and results in much simpler code than that for the simplex algorithm. In practice, we do not include both the S and T arcs for each rule, but eliminate one of them by the mathematical device of subtracting $\min(e_f(x), e_b(x))$ from both their capacities. These diminished capacities will lead to a lower value for the maximum flow (or minimum cut), but will give the same strategy as before, and the optimisation will usually run faster, because fewer arcs must be considered.

This approach can also be modified to allow for storage costs being charged only to those forward rules having a backwards successor. To take account of storage, we must add some more nodes and arcs to the network already described: new nodes representing the facts and goals, and new arcs whose capacities are storage costs. More precisely, for each pattern in F or G we create a node. For each fact or rule pattern we create an extra dummy node, with an arc from the dummy to the real node whose capacity is the cost of storing the relevant set of facts. Finally we add infinite-capacity arcs from the fact nodes to T, from S to the goal nodes, and from every real node to the dummy nodes of its predecessors. Any finite-capacity cut, as described above, will include the arc from a dummy node x' to its real node x precisely when some successor of x is reachable from S (i.e. done backwards) and x is connected to T (done forwards). Thus the capacity of the cut faithfully represents the cost of the corresponding set of choices for rule directions in the way described above, and the minimal cut will give the optimal choices. See Figure 7, where dummy nodes are shown as open circles, and as before, heavy lines represent arcs with infinite capacity.

The network flow algorithm has not been modified to deal with bounds on space or time, because it does not include a way to separate different components of the cost of a strategy. Given that the minimal cost is found from the maximal flow, there seems to be no clear correspondence between strategy costs and flows in the network.

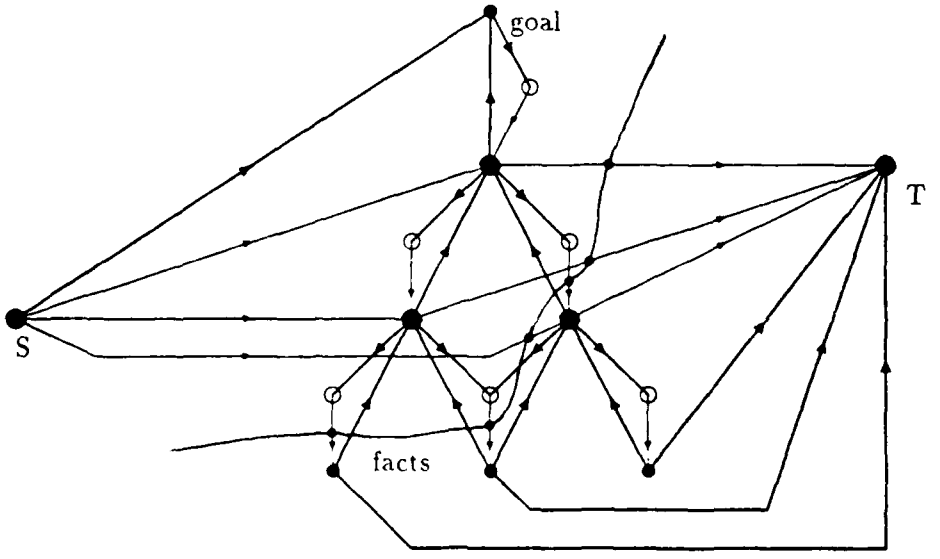


Figure 7: Network with three rules and storage cost nodes

4.2 Duplicate Answers Present

Some rules can generate duplicate answers to a goal, corresponding to different values of some variable which appears in the rule's input literals but not in its output literal. For example, given a rule like $A(x,y)$ & $B(y,z) \Rightarrow C(x,z)$, if facts $A(2,1), B(1,4), A(2,3)$, and $B(3,4)$ were available, then $C(2,4)$ could be deduced twice, once with $y = 1$ and once with $y = 3$. This was discussed more fully in Section 3.1.5. If the rule's conclusions are being stored in the database, the duplicates will disappear, but if the rule is being used in backwards inference and its conclusions forgotten as soon as they are used, then duplicates will not even be detected.

So if some rule r_1 has a predecessor r_2 which generates duplicates, the number of inputs supplied to r_1 will depend on whether r_2 is used forwards or backwards, and this clearly affects the cost of using r_1 backwards. A rule used forwards in a coherent strategy can

receive no duplicates from its predecessors, since duplicates are always eliminated when a rule is used forwards). As mentioned in Section 3.1.5, the cost of using other rules which supply inputs to r_1 can also be affected by this. So if r_3 is such a rule, the cost of using r_3 backwards will depend on $v(r_2)$, but the cost of using r_3 forwards will not. This makes it impossible to express the cost of r_3 as a linear function of $v(r_3)$ and $v(r_2)$, since the value of $v(r_2)$ affects it only when $v(r_3) = 0$; terms containing the product of the two indicator variables would be required to express this.

In fact, the task of optimally choosing the $v(x)$ under these conditions can be shown to be NP-complete, so that it cannot be solved by linear programming unless P=NP.

4.3 The NP-Completeness Proof

We prove that the problem of finding the optimal coherent strategy in the presence of duplicate answers is NP-complete, even if estimating the costs for the rules is in P. The proof is by reduction from the Vertex Cover problem [GJ79, page 46], that of finding a smallest set of vertices of an undirected graph such that every edge of the graph is incident to a vertex in the set; this problem is known to be NP-complete.

We show how to reduce any graph to a triple (F, R, G), such that the optimal coherent strategy for this problem corresponds to a minimal vertex cover. This tells us that an upper bound on the time needed to find a vertex cover is given by the time needed to find the optimal strategy (plus the costs of reducing the graph to the logic program, and of transforming the strategy to the vertex cover; these costs will be shown to be slight). To prove strategy optimisation as a whole NP-complete, we would also need to show that the cost of a strategy can be estimated in time polynomial in the size of the input data; this has been discussed in Section 3.4.

4.3.1 The problem reduction

For each vertex v of the graph, create a predicate $P_v(x)$ of one argument, and for each edge from v to another vertex w, create a predicate $P_{vw}(x)$, but only one such predicate for each edge, i.e. either P_{vw} or P_{wv} but not both. The idea is to arrange things so that to deduce P_{vw} cheaply we need to deduce at least one of P_v and P_w forwards, and if either of P_v or P_w has been deduced forwards, P_{vw} can be deduced backwards at low additional cost. The costs will be such that the optimal R_f must contain, for every P_{vw}, at least one of P_v and P_w, but also contains a minimal number of such P_v. We shall show how to construct R, F, and G so that this holds.

Define R to contain a rule

$$Q_{vw}(x) \;\&\; P_v(x) \;\&\; P_w(x) \Rightarrow P_{vw}(x)$$

for each edge vw, and for each vertex v a rule

$$P_{v1}(x, y) \;\&\; P_{v2}(y) \Rightarrow P_v(x).$$

The goal set G requires us to find all answers to all of the $P_{vw}(x)$.

In all cases, argument x ranges over a domain of values having size d_1, and y has a domain of size d_2. The fact set F contains, for all v and w, the clause sets $(Q_{vw}(X), n_1)$, $(P_{v1}(X, Y), n_2))$, $(P_{v2}(Y), n_3)$. Rather than pull values for the d_i and n_i out of a hat now, we shall develop the constraints on them which are needed for the proof to work, and then give a set of values which satisfy them. Consistency requires that

$$n_1 \leq d_1 \tag{9}$$

$$n_2 \leq d_1 d_2 \tag{10}$$

$$n_3 \leq d_2 \tag{11}$$

Let the graph have V vertices and E edges; we shall assume $E > 1$, otherwise the problem is trivial. We need to show that the numbers of bits needed to encode the graph

itself and the (F, R, G) triple defining the logic program optimisation problem are related by some polynomial. The number of distinct predicates is $3V + 2E$; the number of rules in R is just equal to the sum of the numbers V and E of vertices and edges in the graph, and the number of bits required to represent each rule is bounded above by some expression linear in $\log V + \log E$ (since each literal has to have enough bits to identify its predicate symbol); each predicate appears at most once in F and in G, and the number of bits of information attached to the predicate (describing the domain sizes of its variables, the number of answers to it that are in F, etc.) will turn out to be a linear function of $\lg E$ (the numbers will be powers of E), so that the total number of bits used in this way is a linear function of $E \lg E$.

The transformation of an optimal strategy, once found, into a vertex cover is simplicity itself: for each vertex v of the original graph, check whether the corresponding P_v is in R_f, and include v in the cover iff this is so. This clearly does not take exponential time.

In the next section, we give expressions for the costs of using each rule under different sets of assumptions about the rules adjacent to it in the rule graph. In the one after, we explain why the optimal strategy given these costs will yield a minimal vertex cover.

4.3.2 The costs of the rules

In what follows, we shall write G_i for the cost of generating a single resolvent with i literals in it, S_i for the cost of storing it, and I_j and L_j for the costs of indexing on, and returning one match to, a pattern literal with j arguments. We shall make the reasonable assumptions that $G_i \leq G_{i+1}$, $S_i \leq S_{i+1}$, $I_i \leq I_{i+1}$, and $L_i \leq L_{i+1}$ for every sensible value of i. The equal frequency assumption is used to estimate unification probabilities.

In the derivation of P_v from P_{v1} and P_{v2}, for each value of X there will be n_2/d_1 facts $P_{v1}(X, Y)$, giving $n_2 n_3/d_1$ attempts at unification between literals $\neg P_{v2}(Y)$ and $P_{v2}(Z)$. Each such unification has probability $1/d_2$ of succeeding, so the probability of proving $P_v(X)$ for this value of X will be roughly $1 - e^m$ where $m = n_2 n_3/d_1 d_2$. Write β for $1 - e^m$.

Using forward inference, the cost to actually generate the answers to $P_v(x)$ will include

$I_2 + n_2 L_2$	to look up answers to P_{v1}
$n_2 G_2$	to generate the resolvents
$n_2(I_1 + n_3 L_1/d_2)$	to check if the relevant $P_{v2}(y)$ is true
$n_2 n_3 G_1/d_2$	to generate the answers themselves
$n_2 n_3 S_1/d_2$	to store them, for a total of

$$c_1 = I_2 + n_2(L_2 + G_2 + I_1) + \frac{n_2 n_3}{d_2}(L_1 + G_1 + S_1)$$

If neither of P_v and P_w has been deduced forwards, then the cost of deducing P_{vw} backwards will be

$I_1 + L_1 + G_3$	to look up the rule for P_{vw} and apply it
$I_1 + n_1 L_1$	to look up the answers to $Q_{vw}(x)$
$n_1 G_2$	to generate resolvents $\neg P_v(X) \neg P_w(X) P_{vw}(X)$
$n_1(I_1 + L_1 + G_3)$	to look up and apply the rule for P_v
$n_1(I_2 + n_2 L_2/d_1)$	to look up the answers to P_{v1}
$n_1 n_2 G_2/d_1$	to generate the resolvents
$n_1 n_2(I_1 + n_3 L_1/d_2)/d_1$	to look up the P_{v2} facts
$n_1 n_2 n_3 G_1/d_1 d_2$	to generate the resolvents $\neg P_w(X) P_{vw}(X)$
$n_1 n_2 n_3(I_1 + L_1 + G_2)/d_1 d_2$	to look up and apply the rule for $P_w(x)$
$n_1 n_2 n_3(I_2 + n_2 L_2/d_1)/d_1 d_2$	to look up the answers to P_{w1}
$n_1 n_2^2 n_3 G_1/d_1^2 d_2$	to generate the resolvents
$n_1 n_2^2 n_3(I_1 + n_3 L_1/d_2)/d_1^2 d_2$	to look up the P_{w2} facts
$n_1 n_2^2 n_3^2 G_0/d_1^2 d_2^2$	to generate the resolvents

Write this as

$$c_2 = c_{21} + c_{22} + c_{23}$$

where

$$c_{21} = I_1 + L_1 + G_3 + I_1 + n_1(L_1 + G_2)$$

$$c_{22} = n_1(I_1 + L_1 + G_3 + I_2) + \frac{n_1 n_2}{d_1}(L_2 + G_2 + I_1) + \frac{n_1 n_2 n_3}{d_1 d_2}(L_1 + G_1)$$

$$c_{23} = \frac{n_1 n_2 n_3}{d_1 d_2}(I_1 + L_1 + G_2 + I_2) + \frac{n_1 n_2^2 n_3}{d_1^2 d_2}(L_2 + G_1 + I_1) + \frac{n_1 n_2^2 n_3^2}{d_1^2 d_2^2}(L_1 + G_0)$$

Here c_{21} represents the cost of applying the rule for P_{vw} and looking up the Q_{vw} facts, c_{22} is the cost of finding the answers for P_v, and c_{23} the same for P_w. Notice that

$$c_{22} > \frac{d_1 d_2}{n_2 n_3} c_{23} \tag{12}$$

If P_w (but not P_v) had been deduced forwards, c_{23} would be replaced by

$$\frac{n_1 n_2 n_3}{d_1 d_2}(I_1 + \beta L_1 + \beta G_0)$$

since we expect with probability β that $P_w(X)$ would in fact be in the database. So the cost would be

$$c_3 = c_{21} + c_{22} + \frac{n_1 n_2 n_3}{d_1 d_2}(I_1 + \beta L_1 + \beta G_0)$$

If only P_v had been deduced forwards, c_{22} would be replaced by $n_1(I_1 + \beta L_1 + \beta G_1)$ and the c_{23} term would be multiplied by $\beta d_1 d_2 / n_2 n_3$. Then the cost would be

$$c_4 = c_{21} + n_1(I_1 + \beta L_1 + \beta G_1) + \beta \frac{d_1 d_2}{n_2 n_3} c_{23}$$

Note that the last term of this is less than c_{22}, by (12) above.

If both P_v and P_w had been deduced forwards, the sum would collapse to

$$c_5 = c_{21} + n_1(I_1 + \beta(L_1 + G_1 + I_1) + \beta^2(L_1 + G_0))$$

4.3.3 Consequences for the optimal strategy

If $c_1 + c_3 < c_2$ and $c_1 + c_4 < c_2$ then for every edge vw in the graph it will be advantageous to deduce one of P_v and P_w by forward inference, so the optimal strategy will certainly correspond to some vertex cover of the graph. To show that it corresponds to a *minimal* vertex cover, we need to show that it would not be advantageous to deduce some extra P_v

forwards, at a cost of c_1, and reap savings of $c_3 - c_5$ or $c_4 - c_5$ on all the P_{vw} corresponding to the edges incident to that vertex v. Since there certainly could not be more than E such edges, it will suffice to arrange that $c_1 > E(c_3 - c_5)$ and $c_1 > E(c_4 - c_5)$. So set

$$\begin{aligned} n_1 &= E \\ n_2 &= E^5 \\ n_3 &= E^2 \\ d_1 &= E^3 \\ d_2 &= E^3 \end{aligned}$$

which is consistent with the constraints stated at (9, 10, 11). Then c_1 is a polynomial of degree 5 in E, while the leading term in c_3 is that in $n_1 n_2/d_1$, which is of degree 3 in E. Hence $c_1 > Ec_3$ for sufficiently large values of E, so that certainly $c_1 > E(c_3 - c_5)$. The leading term in c_4 is also proportional to E^3, so that for large enough E, $c_1 > E(c_4 - c_5)$.

Now

$$\begin{aligned} c_2 - c_3 &= c_{23} - \frac{n_1 n_2 n_3}{d_1 d_2}(I_1 + \beta L_1 + \beta G_0) \\ &> \frac{n_1 n_2 n_3}{d_1 d_2} I_2 + \frac{n_1 n_2^2 n_3}{d_1^2 d_2}(L_2 + G_1 + I_1) + \frac{n_1 n_2^2 n_3^2}{d_1^2 d_2^2}(L_1 + G_0) \end{aligned}$$

which will be greater than c_1 if

$$\frac{n_1 n_2 n_3}{d_1 d_2} > 1 \tag{13}$$

$$\frac{n_1 n_2^2 n_3}{d_1^2 d_2} > \frac{L_2 + G_2 + I_1}{L_2 + G_1 + I_1} \tag{14}$$

$$\frac{n_1 n_2^2 n_3^2}{d_1^2 d_2^2} > \frac{L_1 + G_1 + S_1}{L_1 + G_0} \tag{15}$$

and

$$\begin{aligned} c_2 - c_4 &= c_{22} - n_1(I_1 + \beta L_1 + \beta G_1) + \frac{m - \beta}{m} c_{23} \\ &> n_1 I_2 + \frac{m - 1}{m} c_{23} \end{aligned}$$

and so we require

$$n_1 > 1 \tag{16}$$

$$\frac{(m-1)}{m}\frac{n_1 n_2^2 n_3}{d_1^2 d_2} > \frac{L_2+G_2+I_1}{L_2+G_1+I_1} \tag{17}$$

$$\frac{(m-1)}{m}\frac{n_1 n_2^2 n_3^2}{d_1^2 d_2^2} > \frac{L_1+G_1+S_1}{L_1+G_0} \tag{18}$$

Taking these together with $m > 1$ they dominate the previous three inequalities. The values given for the n_i and d_i in fact imply $m = E$, so that we need

$$E > 1$$

$$(E-1)E^3 > \max\left(\frac{L_2+G_2+I_1}{L_2+G_1+I_1}, \frac{L_1+G_1+S_1}{L_1+G_0}\right)$$

which can clearly be satisfied by a large enough value for E.

4.3.4 Q.E.D.

So for any specific deductive system, we can choose an E such that, for every graph with more than E edges, the Vertex Cover problem for this graph is reducible to a problem of finding the optimal coherent strategy for a certain logic program running on this deductive system.

Note that this proof did not make use of the coherence assumption, and so applies equally to the problem of finding the optimal incoherent strategy in the presence of duplicate answers. This problem is in fact NP-complete even if there are no duplicates; we prove this, and give techniques for solving the incoherent case, in Chapter 6.

4.4 Coping With Duplicates

This NP-completeness result is not as discouraging as it may seem to be. Considerable care was required to construct the logic program used in the proof, and we anticipate that most programs encountered in practice would not display the features that make it necessary

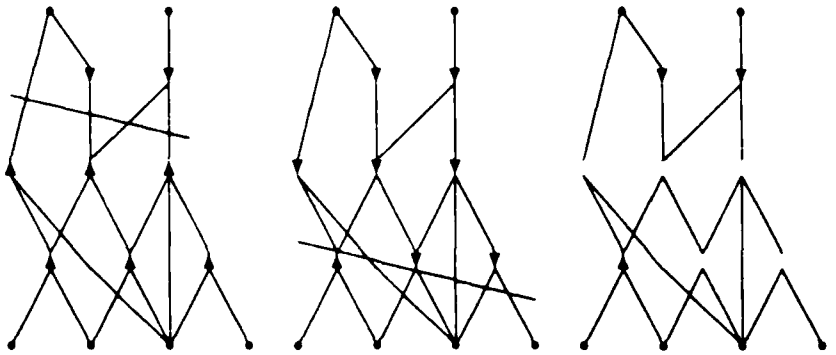


Figure 8: Two strategies which disagree

to explore many strategies. We therefore expect that a heuristic-guided search, endowed with the ability to discard obviously inferior strategies, could solve most such optimisation problems quite fast.

We can still, for any rule, obtain lower and upper bounds on the cost of using it backwards, by assuming either that all duplicates generated by other rules have been eliminated, or that none of them have. Clearly, if we run the linear programming algorithm using optimistic estimates for the costs of using rules backwards, i.e. estimates lower than the true costs, the strategy may mistakenly choose to use some rules backwards that should really be forwards, but will not make the reverse mistake, so we will get for R_f a subset of the truly optimal R_f. If we use pessimistic estimates for backwards costs, we will get a superset. See Figure 8.

If either of these set inclusions is proper, so that these two strategies are not identical, we can remove from R those rules which they agree to use forwards, replace F with the sets of facts deduced by these rules, and re-compute the optimistic and pessimistic cost estimates for the remaining rules. Note that we cannot remove the rules which are agreed to be used backwards (and insert a revised set of goals), because the number of subgoals generated by the backwards rules may depend on whether duplicates are generated by the

rules in the middle. However, now that some rules are definitely used forwards and generate no duplicates, while others are used backwards and will generate duplicates, there is less scope for optimism or pessimism about how many duplicates could occur, so the upper and lower bounds on costs can be tightened, which may give us better values for R_f if we repeat this procedure. Should this approach fail to converge on an agreed strategy, then an exhaustive search must be done, but can still be pruned.

The search algorithm to be outlined here is based on the A^* search described in [Nil80]. It assumes that we have a lower bound for the cost of any completion of a given partial strategy. This bound is obtained simply by using the (unique) estimated costs for forward inference, the optimistic estimated costs for backward inference subject to the duplicates generated by rules which the artial strategy includes as backwards, and then running the linear program.

The search uses as its main data structure a list of partial strategies, sorted by their lower bound cost estimates. It starts with only the empty strategy, in which no directions have been chosen, and runs the linear program on optimistic and pessimistic estimates (as described above) some number of times - in the current implementation, once. As its main iterative step, it will remove from the list and expand (i.e. add another rule to) the strategy with the cheapest lower bound. It generates two new partial strategies, one in which a previously undecided rule and its predecessors are used forwards, the other with it and its successors used backwards. These are not yet added to the list.

In the first of the new strategies, there may be fewer duplicates generated than before, so the pessimistic estimates for the costs of using rules backwards may decrease. We re-calculate these estimates and run a linear program again to see if any more rules are definitely used backwards. In the second new strategy, there may be more duplicates, so that the previous optimistic estimates may increase; this could give us more rules that are definitely used forwards, when we apply the linear program. In either case, if any more rule directons have been fixed, we must now re-calculate the lower bound on strategy cost.

This allows us to insert the new partial strategies in their correct places in the list of partial strategies.

This continues until the cheapest strategy on the list is a complete strategy. The efficiency of the search will depend strongly on how realistic the lower bound estimate is, especially in the later stages. Pearl [Pea83] has shown that the expected time complexity of an A^* search increases like the exponential of the proportional error in the heuristic estimate used. So it seems good to expand a partial strategy by adding the rule with the highest ratio of duplicate answers to unique answers. Such rules can be expected to cause a large difference between optimistic and pessimistic cost estimates, so that once directions have been chosen for them, the two sets of estimates can be expected to bracket the truth more closely. Indeed, once a partial strategy specifies directions for all the rules that can generate duplicates, the linear program will immediately give the optimal completion of this partial strategy, which can be added to the list as a complete strategy.

Alternatively, if some departure from optimality is acceptable, we may halt the search as soon as some partial strategy is found with a pessimistic estimate that is acceptably low. It can easily happen that a rule is theoretically capable of generating duplicate answers, but in fact generates very few duplicates (perhaps because it generates very few answers of any kind), and so the pessimistic and optimistic estimates for strategies using this rule in different directions may be close together. Much effort may be wasted trying to decide whether those few duplicates should be eliminated or not.

The search algorithm can easily deal with time or space bounds, by simply discarding any strategy whose optimistic estimates violate a bound. Roussopoulos points out that, if the bound imposed is very tight, it can be more efficient to generate strategies which satisfy the bound and then estimate the cost of each of them; we shall not examine this approach any further here.

Chapter 5

Ordering Input Literals

In this chapter we remove the assumption that the order of the input literals in a rule is fixed when the rule is given to the optimiser. We show that, even when strategies are required to be coherent, finding both the best direction and the best input literal ordering for every rule is hard.

5.1 Literal Ordering with Equal Expenses

5.1.1 Literal Ordering is NP-complete

Even the simplest case of finding the optimal order for input literals, in which there is only one rule, its direction is known, the goal set consists of only one goal, which is known, and all the inputs to the rule are available by lookup at cost 1 unit per fact, still yields an NP-complete problem.

A full treatment of this case is given in [SG85] and will not be reproduced here. Some of the notation, however, will be carried over. The i'th input literal of a rule (under some ordering, usually written t) will be denoted by p_i, and the sequence of the first $i-1$ literals by $t_{1,i-1}$; the number of answers to a sequence t of input literals, interpreted as a conjunction of subgoals, will be written $Numsol(t)$, and the average number of answers to

p_i given a solution to $t_{1,i-1}$ (namely the average over all the sets of variable bindings that solve $t_{1,i-1}$) will be written $AvgNumsol(p_i, t_{1,i-1})$. In the terminology of Section 3.1.1, this should actually be described as the average number of answers to the mode of p_i obtained by binding all the variables that appear in $t_{1,i-1}$. It is still necessary to take an average, because these variables might be bound to constants, or turned into bound variables, by different sets of answers to $t_{1,i-1}$. Write M for the number of input literals. The following are easily derived:

$$\begin{aligned} Numsol(t) &= Numsol(t_{1,i-1}) \times AvgNumsol(t_{i,M}, t_{1,i-1}) \\ &= \prod_{i=1}^{M} AvgNumsol(p_i, t_{1,i-1}) \end{aligned} \quad (19$$

It then follows that, if the expense of a literal p is equal to $Numsol(p)$, the cost of solving t as a conjunctive goal is given by

$$Cost(t) = \sum_{i=1}^{M} Numsol(t_{1,i})$$

To show that the problem of finding the best sequence t for a given set $\{p_i\}$ of input literals is NP-complete, we need to make some assumptions about $AvgNumsol(p_i, t_{1,i-1})$. We shall adopt the argument independence assumption fron Chapter 3, so that

$$AvgNumsol(p_i, t_{1,i-1}) = Numsol(p_i) \times \prod_{v} U(v)$$

where $U(v)$ is the unification probability associated with a variable v, and the product is taken over all variables v that occur both in p_i and in $t_{1,i-1}$.

The NP-complete problem which we reduce to this problem is "Minimum Cut Linear Arrangement", found in [GJ79, page 201]. This problem is stated as follows: given a graph G with n vertices, and an integer K, find whether there is an ordering of the vertices (with the ordinal number of a vertex v denoted by $f(v)$) such that, for every i with $1 < i < n$, the number of edges (u, v) of the graph such that $f(u) \leq i < f(v)$ is at most K. Suppose we are given an instance of this problem. Reduce it to a literal ordering problem as follows:

Without loss of generality, there is no edge connecting a node to itself. For each edge in G, create a variable, and for each vertex, a predicate symbol. The term corresponding to each vertex consists just of its predicate symbol and the variables of the edges that are incident to it. If a vertex has degree d, let the $Numsol$ of its predicate symbol be n^d, where n is the number of vertices in the graph. For all variables v let $U(v)$ be n^{-2}. We shall prove that there exists an ordering of the query with cost less than n^{K+1} iff there exists a linear arrangement of the kind demanded. If we could find the optimal literal ordering in polynomial time then we could certainly determine whether an ordering with cost less than n^{K+1} existed, also in polynomial time.

First we shall observe that, for a given ordering of the vertices in the graph (terms in the query), the number of edges (u, v) in the graph such that $f(u) \leq i < f(v)$, which we shall call $Cut(i)$, is precisely the number of variables which appear just once in $t_{1,i}$. This is a consequence of the problem reduction, because every variable appears exactly twice in the query (each arc has exactly two endpoints). Also let $W(i)$ be the number of variables that appear twice in $t_{1,i}$. Then $2W(i) + Cut(i)$ is just the number of variable occurrences in $t_{1,i}$, and since the arity of each predicate has been made equal to the degree of the corresponding vertex, $2W(i) + Cut(i)$ is also equal to the sum of the degrees of the first i vertices.

Now

$$Numsol(t_{1,i}) = \prod_{j=1}^{i} Numsol(p_j) \prod_{u} U(u)$$

where the product is over all variables u appearing twice in $t_{1,i}$. Recalling that the $Numsol$ of a predicate is n^d where d is the degree of its corresponding vertex, we see that

$$\prod_{j=1}^{i} Numsol(p_j) = n^{2W(i)+Cut(i)}$$

Therefore, since $U(w) = n^{-2}$ for all w, for any i from 1 to $n-1$,

$$Numsol(t_{1,i}) = n^{Cut(i)}$$

and $Numsol(t_{1,n}) = 1$.

Thus if the vertices of the graph are arranged so that $Cut(i) \leq K$ for all relevant i, the cost of the corresponding ordering of the terms of the query will be at most $1 + (n-1)n^K$ which is less than n^{K+1}, whereas if there is some i such that $Cut(i) > K$, then the cost will be at least n^{K+1}. This is what we wanted.

5.1.2 A literal ordering algorithm

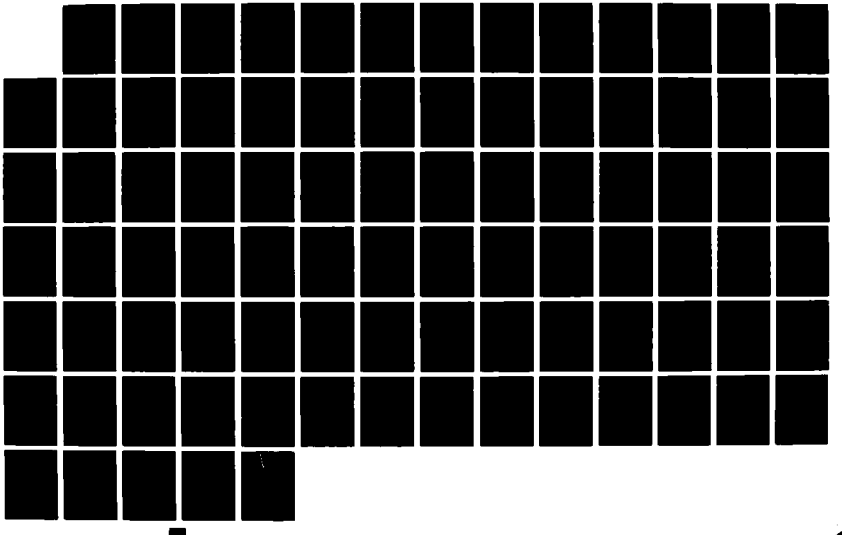
Smith [SG85] has proposed an algorithm for finding the optimal ordering under these circumstances, based on "best-first" search of the space of sequences of subsets of the literals, starting with the empty sequence and generating new ones by adding a literal at a time to the end of the sequence. This search is aided by various features of the problem which allow many nodes in the search space to be pruned. One of the most elegant and powerful of these is his observation that if, for a sequence t of length j,

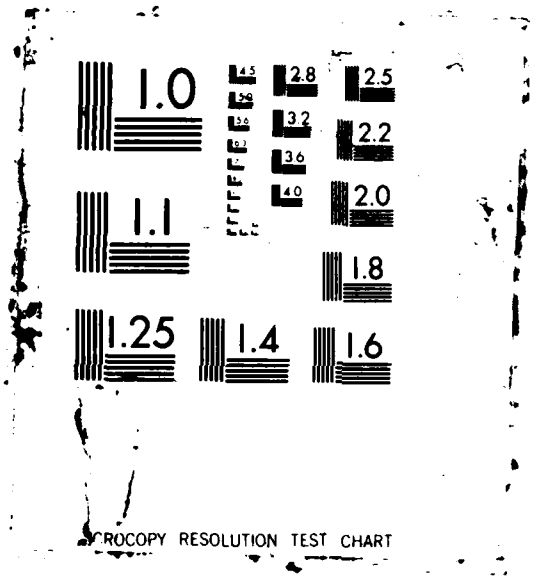
$$AvgNumsol(p_j, t_{1,j-2}) < AvgNumsol(p_{j-1}, t_{1,j-2})$$

then the sequence t' obtained by transposing p_{j-1} and p_j will have lower cost than t, and so sequences that have t as an initial subsequence need not be considered. This is called the Adjacency Restriction; it also means that, if the search algorithm generates a sequence ending in some literal p, the next one to be added to this sequence should be one of those whose $AvgNumsol$ was higher than that of p when p was added. Any literal not satisfying this condition would clearly give a non-optimal sequence if it were added immediately after p. As a corollary, there must be at least one literal satisfying the condition, whence it is not possible that p had the highest $AvgNumsol$ of all literals available when it was chosen. These two constraints alone allow a large proportion of the possible permutations of all the input literals of a rule to be eliminated. In the case of a rule with 8 input literals, 1,385 of $8! = 40,320$ possible permutations are ruled out. Smith also proves that the lowest $AvgNumsol$ of any literal not yet in the sequence is less than that of those that can be added next, with at most a trifling increase over the optimal cost.

AD-A179 493 SEQUENTIALIZATION OF LOGIC PROGRAMS(U) STANFORD UNIV CA 2/2
DEPT OF COMPUTER SCIENCE R J TREITEL NOV 86
STAN-CS-86-1135 N00014-81-K-0303

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
CROCOPY RESOLUTION TEST CHART

Section 5.4.3 below.

All these results apply only to literals whose cost is equal to the number of answers to them (or rather, linearly related, with the same constant of proportionality for all literals). The Adjacency Restriction in fact applies to any pair of literals having the same marginal expense per answer, and the result about the cheapest literal applies if its marginal expense per answer is no higher than that for any other literal remaining. When we are dealing with literals of which some can be looked up in the database while others must be inferred, these conditions will frequently fail to hold, so Smith's algorithm will lose much of its power.

5.2 Literal Ordering with Unequal Expenses

5.2.1 Previous algorithms

Warren [War81] suggests ordering a set of literals by starting with the one that is cheapest to solve, and then repeatedly adding the one that is cheapest given the variable bindings from the the preceding ones. While this yields an ordering very quickly (its running time is $O(n^2)$), it can easily be shown that there are cases where it does not yield the optimal one. Warren claims that in practice it usually yields a good one.

The problem of optimally ordering literals having unequal expenses has been studied by Ibaraki and Kameda [IK84], who present an algorithm for finding the optimal ordering in time $O(n^2 \log n)$, subsequently improved by Krishnamurthy *et al.* [KBZ86] to $O(n^2)$. This algorithm, though, depends on assumptions about both the set of input literals and the final ordering that do not always hold. In particular, it applies only where the graph corresponding to the query, as described in the NP-completeness proof above, is a tree (and has no "hyperedges", i.e. no variable appears in three or more literals). It also looks only at orderings such that p_i always has at least one variable in common with $t_{1,i-1}$, even though Smith points out that the optimal ordering need not have this property.

In fairness, the possible optimality of an ordering not having this property seems (although this has not been proved) to be a consequence of one of our assumptions about indexing, namely that the indexing scheme used is good enough to return only a few candidate answers when given a highly specific query against the database. This underlies the belief that the expense of finding the answers to a literal is equal to the number of the answers, or at least proportional to it with a constant of proportionality that does not vary across literals or modes. Since real-world databases, other than those used to store facts for a logic program, generally do not index a relation on all possible attributes, it is unlikely that they would satisfy this assumption, and so this restriction on orderings considered may not be serious in the context of a large database.

Still, most rules that have enough input literals for their ordering to be a non-trivial problem do not yield a tree, so that this algorithm is not directly usable. Krishnamurthy suggests that the graph could be coerced into tree form by deleting some edges, which corresponds to pretending that some variables have unification probability equal to 1; another possibility would be to collapse each cycle in the graph to a single node. Each of these could result in serious departures from optimality, though the edge deletion method can prevent generation of absurdly bad orderings, while the other might fail even to do this, if the graph was one big cycle.

5.2.2 An admissible algorithm

A generalisation of Smith's Adjacency Restriction remains true, namely that for any t and t' which contain the same set of literals, only the cheaper of the two need be retained and expanded further: any sequence beginning with t can be turned into a cheaper one by replacing t with t' and leaving the rest unchanged, if t' is indeed cheaper than t. This can be used to prove a version of the Restriction that does hold between literals having different expenses per answer. However, the corollary of the Restriction that was most valuable in limiting the search space depended on the expense per answer being the same

across different modes of the same literal, since this made it particularly cheap to check which literals could appear next in the sequence: it was only necessary to look at their *AvgNumsol* values given the current sequence, and these values were already known. This happy property is not guaranteed when answers are obtained by inference, and so each attempt to restrict the set of literals that can appear in a given position requires another calculation of expenses. The power of the tree query requirement is due precisely to the limit it places on the number of distinct modes of a literal that can arise in an ordering, and hence on the number of different expenses that must be found and considered.

We may take advantage of this generalisation of the Adjacency Restriction by keeping track of which sets of literals have appeared, and the costs of the best known permutation of each such set. The number of sets is at most 2^n, where n is the number of input literals, but is usually much less, because in non-pathological cases the search will generate only the best few orderings and a few worse ones. Thus the cost of storing the best cost known so far for each set of literals encountered should not be prohibitive. To find (or fail to find) some sequence among these sets would require at most roughly $\log(2^n) = n$ comparisons of sets, each of which would take up to n comparisons of literals, for a product of at most n^2 literal comparisons. So the overhead cost of this approach is by no means unreasonable if it leads to a significant reduction in the number of sequences generated.

There seems to be no guarantee that the search will not generate many bad permutations in decreasing order of cost, so that this modification to the algorithm may have poor worst-case performance. To strengthen it we can use the approximate algorithms of Warren or Krishnamurthy as "filters" on sequences generated: for every sequence t that is not found in the table of sets of literals already seen, the set of literals in it can be submitted to the approximate algorithm, and if the permutation returned is cheaper than that in t, it should be used and t should be dropped. Either way, the cheaper of the two can be entered in the table, since finding a set in this table takes no longer than running either algorithm.

In fact, to the extent that these algorithms involve computing the expenses of literals, they will take longer than finding a set in the table. So the table should be used as much as possible.

Since the best-first search will initially generate sequences consisting of cheap literals, and only gradually consider more expensive ones as it is forced to do so, the benefits of using the cheapest-first algorithm as a filter are uncertain, and the tree query algorithm is probably preferable for this purpose. Since estimating the expense of a literal is likely to take much longer than looking up sets of literals, it seems best to do this filtering only rarely, such as when a literal that was not in any previously seen sequence has just been added. This would take care of the main weakness of the best-first algorithm, which is that it will postpone adding an expensive literal until it has to, although it might in fact be better to "grasp the nettle" early.

We shall take the liberty of describing the best-first search augmented by this filtering mechanism as the "modified Smith" algorithm. Smith notes the generalised restriction, but does not pursue it very far.

The subset test

Provided the argument independence assumption (see Section 3.1) holds, and that no constant appears both in some literal of t' and in one of the rules or facts that could be used to solve that literal, we can replace the test for set equality between t and t' by a test for containment: if the set of literals in t is a subset of that in t' and the estimated cost of t is greater than that of t', then t can be discarded. This requires a lemma, namely that under the two assumptions mentioned, the expense per answer of a literal does not decrease when some of its arguments change from free to bound variables or from bound variables to constants, and its total expense does not increase.

The proof of the lemma is easy for changing from free to bound variables. Without loss of generality, suppose that there is only one variable to be changed from free to bound.

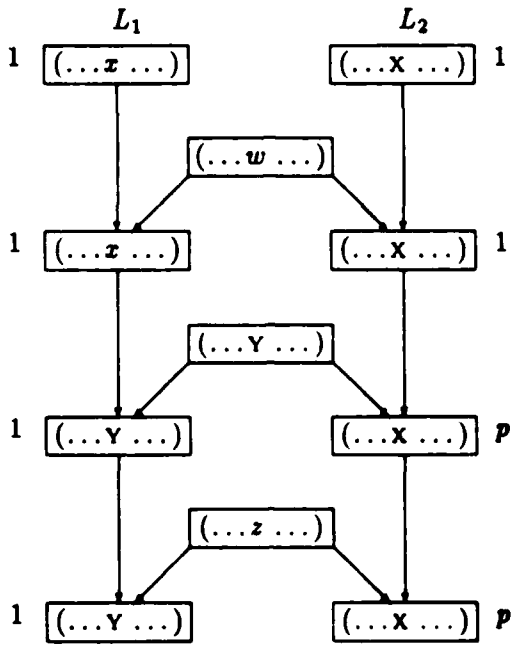


Figure 9: Bound versus free variable

Consider the clause sets generated by the simulator in estimating the expense of the literal L_1 with the free variable, and those that would be generated by doing the same resolutions starting with a literal L_2 that was identical to L_1 except in having one of L_1's free variables bound. The set of answers to L_2 would clearly be smaller by a factor p, where p is the unification probability associated with this bound variable, since it would have had to unify at some point with a bound variable of some fact pattern. Of the other clause sets generated on the path to the answer, those coming before this resolution will have the same size in each case, and those after will be smaller if the variable was bound initially. So the expense for L_2 will be at least p times the expense for L_1, and the number of answers will be exactly p times as much, whence the expense per answer cannot decrease. This is illustrated in Figure 9.

It is now clear why the assumption about constants was needed: a constant appearing

twice in such a fashion could unify with itself in the simulation, giving a higher unification probability than if it had been unified with a bound variable. But if it can only unify with variables (bound or free) and with different constants, then the unification probability will either be zero or else equal to that for a bound variable, and the same argument is valid for changing a bound variable to a constant.

Note also that, in either of these cases, the more bound literal has lower (or no higher) expense than the less bound one, since it takes part in fewer (or no more) resolutions. This is the reverse of the result on expense per answer.

Before proving the validity of discarding a sequence that is more expensive than a superset of itself, we observe that Smith's expression for the cost of a sequence must now be re-written to take account of different expenses for different literals (and for different modes of the same literal), and this can be done in two ways. Instead of

$$Cost(t) = \sum_{i=1}^{M} Numsol(t_{1,i})$$

we have

$$Cost(t) = \sum_{i=1}^{M} ExpAns(p_i, t_{1,i-1}) Numsol(t_{1,i}) \tag{20}$$

$$Cost(t) = \sum_{i=1}^{M} Numsol(t_{1,i-1}) Expense(p_i, t_{1,i-1}) \tag{21}$$

where $Expense(p_i, t_{1,i-1})$ is the expense of solving one instance of p_i subject to the variable bindings caused by solving $t_{1,i-1}$, and $ExpAns(p_i, t_{1,i-1})$ is this expense divided by $AvgNumsol(p_i, t_{1,i-1})$.

Suppose now that t has higher cost than t' but contains a subset of its literals; from the assumption that t is an initial subsequence of some optimal sequence we shall derive a contradiction. Without loss of generality, t' need only contain one more literal than t (if not, just delete the last few literals of t', which can only reduce its cost). Clearly this extra literal (call it l) is not the last literal of t', for then t' would be an extension of t and would cost more than it. Consider the sequence t'_l obtained by removing l from t'. This sequence

is a permutation of t, and so, under the assumption that t begins an optimal sequence, t'_l must have cost at least as high as t, whence its cost is higher than that of t'. Thus removing l from t' must have increased the total expenses of some of the literals of t' that came after l. Since removing l can only have caused these literals to become less bound, it cannot have increased the expense per answer of any of them, so in order to increase the total cost, it must have increased the $Numsol$ factor in some of the terms of $Cost(t')$ as calculated by (20), whence the $AvgNumsol$ of l in the position it occupied in t' was less than 1. So t'_l has more answers than t', since using (19) we see that removing l will have increased the product of the $AvgNumsol$ values, and unbinding the variables that were bound by l cannot have reduced any of these values. But since t'_l is a permutation of t, it follows that t has more answers than t'.

Now consider an optimal sequence beginning with t; let u be the subsequence of it between the last literal of t and the place where l appears, and v the rest after l, so this optimal sequence can be written $t|u|l|v$. Compare it with the sequence $t'|u|v$, using (21). For each literal in v, the corresponding terms are equal in the expressions for $Cost(t|u|l|v)$ and $Cost(t'|u|v)$. The costs of literals in u have a $Numsol$ factor that is lower in $Cost(t'|u|v)$, because $Numsol(t') < Numsol(t)$, and an $Expense$ factor that is no higher, because t' will have given them at least as many variable bindings as t, and the lemma applies. So u contributes less to $Cost(t'|u|v)$ than to $Cost(t|u|l|v)$. And we already know that $Cost(t') < Cost(t)$, whence we get

$$Cost(t|u|l|v) > Cost(t'|u|v) + Numsol(t|u)Expense(l, t|u)$$

This contradicts our assumption that $t|u|l|v$ was optimal.

5.3 Literal Ordering with Known Directions

If we know, or have assumed, that a rule r is used forwards, then by coherence r's predecessors are used forwards too, and so r's inputs are all going to be available in the database

and can be looked up at known expense per answer. So we can apply the Smith algorithm, or the modified one if necessary, and immediately obtain the optimal ordering of r's input literals, valid over all strategies in which r is used forwards.

It is easy to find the optimal literal orderings for all rules given a fixed coherent strategy (set of directions), if we allow one extra trick. This is the provision of different versions of rules, with different input literal orderings, for different modes of goals; the mode of a goal or literal is then determined at run-time and the correct version of each applicable rule chosen. If we can use such "polymorphic" rules, we can find the optimal orderings of the input literals for any set of backwards rules whose inputs are available by lookup, and this can be dovetailed with the above result for forwards rules; it also provides a useful heuristic function to guide an A^* search of the space of coherent strategies with literal re-ordering.

5.3.1 Multiple literal orderings

It may easily happen, especially for rules having more than one successor, that different invocations of the rule have different sets of variables bound, i.e. involve different modes of the rule's output literal. Since the expenses, and expenses per answer returned, of different modes of a literal can vary widely, it is likely that the optimal input literal orderings would vary between invocations of the rule that differ in this way. Any of the above ordering algorithms can use cost estimates that are summed over all invocations of the rule, and return a compromise ordering, but this is likely to lead to higher cost than if a special-purpose ordering were used for each mode of invocation. The compromise ordering would also depend on the estimated "mix" of goals presented to the rule, whereas the special-purpose orderings would not, and so they would not depend on the ways in which other rules were used. Thus the use of multiple orderings has the potential for both improving execution speed and simplifying optimisation.

The run-time overhead of recognising which mode a subgoal is and dispatching it to the correct version of the rule can be reduced by encoding the mode of a literal in the name of

its predicate. Following a conventional practice, this could simply be the original predicate name concatenated with a string, such as "fbb" to indicate that the first argument is free and the second and third are bound. Each rule would now have many versions for backwards use, each using a different mode of its output literal; both the order and the modes of the input literals of such a rule version would be determined by the literal ordering algorithm, and would be optimised for solving goals of the appropriate mode. The literal ordering algorithm would have to return the mode of each input literal as well as its position in the optimal ordering, but this is easy to do.

It would be too user-unfriendly to insist on using new names for the goals in G, so we will still need to examine each top-level goal presented to the system, recognise its mode, and substitute predicate names appropriately. This, though, should be cheap compared with doing the same thing for every clause generated.

5.3.2 Orderings for backwards rules

Armed with this mechanism, we can now determine optimal input literal orderings for the backwards rules in a coherent strategy quite simply. Conceptually, we start with those rules all of whose predecessors are either facts or rules used forwards, so that the expense per answer of their inputs is just that of looking up the answers. The modified Smith algorithm can be used here to find the optimal literal order for every version of each of these rules. Once these orders have been found, the expense per answer of any input literal that unifies with an output literal of one of these rules can be computed, and so we can work on the next "layer" of rules, using the modified Smith algorithm again. Iterating this up the rule graph, we can clearly obtain sets of input literal orderings for all the rules as long as the graph has no cycles.

In practice it would be expensive to do the computation this way, since the number of possible different modes of goals for a rule can be quite large. It will be better to start from the top and work downwards recursively, finding the optimal ordering of the input literals of

a rule r only for those goal modes that arise in the processing of r's successors. Even these modes may turn out to be numerous, since the search for optimal orderings can generate many different modes of a literal.

The literal orderings so produced will only be valid if the cost of using a backward rule to answer some subgoal is independent of how that subgoal was generated, i.e. the cost should depend only on the first literal of the clause that invoked the rule. Otherwise, there might be two different paths through the rule graph from this rule to a top-level goal or a forward rule, resulting in different sets of clauses being generated, so that the expenses of the answers to input literals of the rule might vary. Since this variation could not be predicted, there would be no good data on which to run the modified Smith algorithm. Thus we see that, referring back to Section 3.2, the practice of instantiating resolvent clauses completely not only is expensive at run-time but also leads to difficulties in optimisation, because the cost of processing some clause depends on the length of its tail.

Unfortunately, it is difficult to be sure whether a rule is going to be used forwards or backwards in the optimal strategy. Indeed, the optimal choice of rule directions when input literals can be permuted is NP-complete even in a case where the literal ordering itself is easy, such as where no rule has more than two input literals.

5.3.3 NP-completeness of choosing rule directions

This NP-completeness proof is similar to the one given in Section 4.3, being based on a reduction from the Vertex Cover problem. The reduction, and the correspondence between optimal strategy and minimal vertex cover, are very similar: the minimal vertex cover corresponds to the set of rules used forwards in the optimal strategy.

The rules are

$$P_v(x,y)\ \&\ P_w(y,z) \ \Rightarrow\ P_{vw}(x,y,z)$$

$$P_{v1}(x,y)\ \&\ P_{v2}(x,y) \ \Rightarrow\ P_v(x,y)$$

As before, let the graph whose vertex cover we are trying to find have E edges; we shall show that there is a value of E such that the problem reduction works for all graphs with at least that many edges.

All arguments come from the same domain, of size d (the equal frequency assumption is used). There are n_1 facts for each predicate P_{v1} and n_2 for each P_{v2}, and n_3 goals of the form $P_{vw}(K_1, y, K_2)$ for every P_{vw}, where K_1 and K_2 represent unknown constants. So the number of true instances of each P_v is $n_1 n_2/d^2$, and of each P_{vw} is $n_1^2 n_2^2/d^5$. Departing from tradition, we shall immediately reveal that

$$\begin{aligned} d &= (kE)^3 \\ n_1 &= (kE)^5 \\ n_2 &= (kE)^5 \\ n_3 &= k^4 E^3 \end{aligned}$$

where k is a constant close to 1. Then we get

$$\begin{aligned} N(P_v) &= (kE)^4 \\ N(P_{vw}) &= (kE)^5 \end{aligned}$$

For brevity, we shall use P_v to stand for the predicate which appears first in a P_{vw} rule, even though we are considering strategies which permute the input literals. Likewise P_w will stand for the other predicate. Since P_{v1} and P_{v2} have the same number of answers, it does not matter which of them appears first in the rule for P_v.

The cost of deducing P_v forwards is

$$c_1 = I_2 + n_1(L_3 + G_2 + I_3) + N(P_v)(L_3 + G_1 + S_1)$$

with an additional cost of at most $kEN(P_{vw})L_2$ for looking up the stored answers. This was obtained by considering the most expensive case, in which every one of the P_{vw} rules is used backwards and has this P_v at its second input literal, which is the one where more

lookups are done. The dominant term in the sum of these two is $n_1(L_3 + G_2 + I_3)$, which is proprtional to E^5.

The cost of solving P_v backwards with one variable bound, as would be done if it were the first input literal of a P_{vw} rule used backwards, is

$$c_2 = n_3\left(G_3 + \frac{n_1}{d}(L_3 + G_2 + I_3) + \frac{N(P_v)}{d}L_3\right)$$

for each P_{vw} rule that invokes this P_v rule; the dominant term in $e_8(P_v)$ is thus at least $kn_1(L_3 + G_2 + I_3)$. So provided that $k > 1$, the backwards cost is higher than the forwards cost for large enough E, and the optimal strategy will use forwards any rule whose output literal unifies with the first input literal of one of the P_{vw} rules. This gives us the first half of what we want for the problem reduction: for each edge vw, there is a node v corresponding to a rule used forwards.

The costs of deducing P_{vw} forwards and backwards, ignoring the costs already attributed to P_v and P_w, will then be

$$\begin{aligned} c_3 &= I_2 + N(P_v)(L_2 + G_2 + I_2) + N(P_{vw})(L_2 + G_1 + S_1) \\ c_4 &= n_3\left(I_2 + \frac{N(P_v)}{d}(L_2 + G_1 + I_2 + L_2) + \frac{N(P_{vw})}{d^2}G_0\right) \\ \text{or } c_4' &= n_3\left(I_2 + \frac{N(P_v)}{d}(L_2 + G_1 + I_2) + \frac{N(P_{vw})}{d^2}(L_2 + G_0)\right) \end{aligned}$$

where c_4 is obtained by assuming that P_w is inferred backwards, and c_4' by assuming it to be used forwards. The last term of c_3, representing the number of P_{vw} facts that could be deduced forwards, is proportional to E^5, and so it dominates all the other terms, and all the terms of c_4 and c_4'. So the rule for P_{vw} will be best used backwards.

For inferring P_w backwards with both variables bound we get, for each P_{vw} rule invoking this P_w at its second input literal, a cost of

$$c_5 = n_3\left(\frac{N(P_v)}{d}(G_2 + I_3) + \frac{n_1 N(P_v)}{d^3}(L_3 + G_1 + I_3) + \frac{N(P_{vw})}{d^2}L_3\right)$$

and the dominant term here is $n_3 N(P_v)(G_2 + I_3)/d = k^5 E^4(G_2 + I_3)$, which when multiplied

by E is less than the dominant term of c_1, namely $(kE)^5(L_3 + G_2 + I_3)$, so that P_w will definitely be inferred backwards unless it appears in the first literal of some P_{vw} rule.

This makes it clear that the optimal strategy corresponds to a minimal vertex cover, with the rules that are used forwards corresponding to the nodes that are in the cover. Note that this proof does not need the coherence assumption, and so would also apply to the search for an optimal incoherent strategy with input literal re-ordering. Admittedly the optimal strategy displayed is a coherent one, but this is just because the rules for P_{vw} are more expensive to use forwards, for a reason that is independent of the directions of their predecessors. There are also no duplicates involved.

5.4 Literal Ordering with Changing Directions

The essential point of this NP-completeness argument is that literal ordering depends on knowing the expenses of inputs to a rule, which are determined by (among other things) rule directions, and rule directions are determined by rule costs which depend on literal orderings, so some kind of marriage between literal ordering algorithms and rule direction algorithms is clearly necessary.

5.4.1 Local Improvements

What we would like to be able to do is start with some set of presumed expenses for all literals, obtain an ordering of the input literals for each rule, and then run an algorithm from Chapter 4 to determine the direction of each rule. We could then find the optimal input literal ordering(s) for each rule as explained in the previous section. However, the expenses implicitly determined for each literal by the directions and orderings so found would probably be different than the ones presumed at first, so we should be obliged to re-do the choices of rule directions, and this would lead to new orderings for some of the backwards rules. This would in turn force us to revise the expenses

for input literals again, completing a feedback loop. It is important to know whether this feedback would be positive or negative, i.e. whether a small change would be self-reinforcing or not. If the feedback is negative, then we can iterate this procedure, cyclically revising the expenses, input literal orderings, and rule directions, and have high hopes that we will converge on the optimal strategy. Positive feedback usually causes divergence to some point in the space which has no special properties except that further divergence in the same direction is impossible. Systems with positive feedback usually have several stable states of this kind, and few or none of them are optimal in any way.

Elementary economic theory suggests that the feedback will be positive. If some input becomes cheaper, presumably the literal ordering will change so that more of it will be demanded and used; and this increase in demand can cause it to become cheaper still, by increasing the estimated costs of using some rules backward, so that they are more likely to be switched to forwards (economies of scale). We should, however, justify the use of economics here.

5.4.2 Supply and Demand

Let l be some input literal of r, and assume that the rule(s) deducing answers to l, or some predecessors(s) thereof, have just been changed from backwards to forwards. Let S_1 and S_2 be the old and new optimal orderings of input literals in r, and assume that they are not the same. In fact they could be very different indeed: the search algorithm could end up choosing between two different orders that had almost the same estimated cost, and a slight change in the expense of one literal could make one or other of these orders be the cheaper. Let $Cost(S_1)$ and $Cost'(S_1)$ be the total costs of S_1 under the old and new expenses respectively, and similarly for S_2. Let l_1 and l_2 be the modes of l that are used in S_1 and S_2. Let $N_{S_j}(l_i)$ be the number of goals beginning with literals like l_i that are generated under S_j, and let $Expense(l_i)$ and $Expense'(l_i)$ be old and new expenses.

Then

$$Cost(S_1) - Cost'(S_1) = N_{S_1}(l_1)(Expense(l_1) - Expense'(l_1))$$

$$Cost(S_2) - Cost'(S_2) = N_{S_2}(l_2)(Expense(l_2) - Expense'(l_2))$$

but

$$Cost(S_1) < Cost(S_2)$$

$$Cost'(S_1) > Cost'(S_2)$$

so

$$Cost(S_1) - Cost'(S_1) < Cost(S_2) - Cost'(S_2)$$

whence

$$N_{S_1}(l_1)(Expense(l_1) - Expense'(l_1)) < N_{S_2}(l_2)(Expense(l_2) - Expense'(l_2)) \qquad (22)$$

If $l_1 = l_2$, we get $N_{S_1}(l_1) < N_{S_2}(l_2)$, so the number of l goals increases. Also,

$$Cost(S_1) > Cost'(S_1) \geq Cost'(S_2)$$

(As any economist could tell you, when inputs become cheaper, so do outputs.)

If the $Expense'$ numbers are zero, which happens if all the rules deducing l are switched to forwards, then (22) becomes

$$N_{S_2}(l_2)Expense(l_2) > N_{S_1}(l_1)Expense(l_1)$$

which means that the apparent expense of solving$l$ backwards is now higher than it was before we re-ordered r's input literals, thus reinforcing the incentive to infer l forwards. So, the "repeated local improvement" procedure does not seem to be usable, and some kind of search across the space of possible strategies will be needed.

An example

For a simple example of how the local improvement procedure can get stuck at a local optimum, refer to the NP-completeness proof of Section 5.3.3 and consider a graph having three vertices u, v, w with edges uv and uw. It is clear that the minimal vertex cover is $\{u\}$, so that the optimal strategy will use only the rule for P_u forwards, and will order the input literals of the rules for P_{uv} and P_{uw} so that the P_u literal is first. But consider a strategy with the rules for P_{uv} and P_{uw} both ordered the other way. The arguments given in Section 5.3.3 show that, with these orderings, it is best to deduce P_{uv} and P_{uw} backwards, P_v and P_w forwards, and P_u backwards. In the notation of Section 5.3.3, we have $E = 2$; let us set $k = 2$ so that

$$\begin{aligned} d &= 64 \\ n_1 &= 1024 \\ n_2 &= 1024 \\ n_3 &= 128 \\ N(P_v) &= 256 \\ N(P_{vw}) &= 1024 \end{aligned}$$

Assuming that all the I_i, L_i, G_i, and S_i are 1, we get

$$\begin{aligned} c_1 &= 1 + 3 \times 1024 + 3 \times 256 = 3841 \\ c_2 &= 128 \times (1 + 3 \times 1024/64 + 256/64) = 6784 \\ c_5 &= 128 \times (2 \times 256/64 + 3 \times 1 + 1/4) = 1440 \\ c_4 &= 128 \times (1 + 4 \times 256/64 + 1/4) = 2208 \\ c_4' &= 128 \times (1 + 3 \times 256/64 + 2 \times 1/4) = 1728 \end{aligned}$$

The strategy described has a cost of $2c_1 + 2c_4 + 2c_5 = 14978$. If the input literal ordering of one or both rules for P_{uv} or P_{uw} were changed, without changing any rule directions, there

would be a cost of at least $2c_1 + c_2 + 2c'_4 = 17922$. So with these directions, the orderings given are the best. Thus, if the local improvement algorithm were to arrive at this strategy, it would find a local optimum which is clearly not globally optimal (the optimal strategy costs only $c_1 + 2c_4 + 2c_5 = 11137$).

5.4.3 Lower bounds for partial strategies

As before, the centrepiece of our search algorithm is a function that gives a lower bound on the cost of any completion of a partial strategy. Since we have seen that the optimal literal orderings can be determined given a complete set of choices for rule directions, we shall confine ourselves to searching in the space of sets of directions, and expect to obtain literal orderings as a side-effect of this.

It is also true that given a complete set of input literal orderings, we can determine the e_f and e_b of each rule, modulo the presence of duplicate answers, and thereby find the optimal set of directions, as discussed in Chapter 4. Thus we could search the space of sets of orderings, with directions being found as a side-effect. However, this space is many times larger than the space of sets of directions, and even though good orderings will be scarce compared to the size of the space, it is hard to focus the search towards them. This is because the best order for a rule depends on the directions and input literal orderings of other rules, and this brings us back to the approach via sets of directions.

Given a coherent partial strategy, we can obtain a lower bound on the cost of backwards deduction in any completion of this strategy by using the procedure described above in Section 5.3.2, under the assumption that all rules not mentioned in the partial strategy are used forwards. Should this assumption turn out false, then the cost of the backwards rules will increase, since the expenses of their inputs will rise, so the procedure definitely yields a lower bound. The cost of the rules that are used forwards is, of course, easy to find exactly.

As remarked above, this computation for backwards rules can be slow, because many modes of goals may have to be looked at. The number of such modes can be reduced, at a cost of some loss of accuracy in the lower bound estimates, by ignoring the expenses of literals whose expected number of answers is less than or equal to 1. It seems reasonable to expect that, in most cases, the total expense of getting such a small number of answers will not be very large. If we assume it is negligible, then Smith's result tells us that these literals may as well appear before all the others. The expense of solving a conjunction of literals is then bounded below by the number of solutions ($AvgNumsol$) to the conjunction of these "low-volume" literals times the cost of the optimal ordering of the other ones; the "high-volume" literals are now the only ones whose expenses need to be computed.

The low-volume literals will mostly be ground literals (i.e. those that have no free variables in them by the time they are resolved upon) and literals whose predicates are really functions (i.e. they will have just one free variable, and it is known that only one binding for it will be returned). Most rules will have some input literals of which one or other of these is true, or becomes true after the output literal has been resolved against a goal. If some of the low-volume literals removed are of the second kind, having a free variable, then this variable can now be considered bound, which may enable us to remove more literals.

This can be iterated until quiescence, and the modified Smith algorithm applied to the remaining literals. Note that the set of low-volume literals obtained in this way will be, for rules used backwards, a function of the mode of the literal invoking the rule. This highlights once again the usefulness of having multiple versions of each rule for the different possible invoking modes.

It is in fact not necessary to ignore the low-volume literals entirely. By applying the *unmodified* Smith algorithm to all the input literals (i.e. assuming that the expense of each literal is equal to its number of answers), we obtain a lower bound on the expense of using

the rule backwards. If this is higher than the bound obtained by looking only at the high-volume literals, then we should use it instead.

5.4.4 Searching for a coherent strategy

Now that we know how to find lower bounds on both the forwards and backwards parts of the cost of completions of a partial strategy, it becomes realistic to do an A^* search of the space of such strategies, very much as in Section 4.4. The lower bounds obtained here will include zero costs for rules that are not mentioned in the partial strategy, so that at first they will not be very realistic, but this will improve steadily as more rules are included. The order in which new rules are added is important.

Inaccuracy can be tolerable in the early stages of the search, when the purpose of the lower bound estimate is mainly to prevent (or defer) consideration of partial strategies that can be seen to involve high costs. Later on, the emphasis shifts: presumably a number of fairly good partial strategies have been developed, and the cost estimates must be used to distinguish the best from the good, so accuracy is important. More formally, the nature of the A^* search guarantees that the partial strategies examined are precisely those that can be obtained by adding one rule to a partial strategy whose lower bound cost estimate is less than the eventual estimated cost of the optimal strategy. So if by using a more accurate lower bound estimate we can push the lower bound for some partial strategy above this number, then we have saved some work: if we increase the estimate but it stays below the optimal cost, we have merely postponed consideration of the strategy. An early partial strategy with few rules in it will usually get a small lower bound estimate from any method, so the chance that we can show it to be non-optimal is small; this chance increases as the partial strategies include more rules and the lower bound estimates come closer to the truth.

This means that in the early stages of the search, we will want to use the quicker method for estimating lower bounds on the total expense of backwards computation, since we are

more interested in detecting the worst partial strategies than in sorting out the good from the best. The lower bounds obtained by this method are affected only by rules whose outputs are among the "high-volume" inputs of another rule, so clearly a partial strategy should be augmented by adding one of these rules to it. Eventually we will arrive at partial strategies that include all these "high-volume" rules. We must now add "low-volume" rules, and must do full input literal ordering on their successors; we can continue to use the quick method on rules for which only their high-volume predecessors are mentioned in the strategy. As a strategy approaches completeness, the lower bound estimate for its backward rules will get closer and closer to the truth, and will be exact for a complete strategy.

However, if clauses are completely instantiated at run-time, exact costs will not be obtainable even for a complete strategy, because the conjunct ordering algorithm will have inexact inputs. Lower bounds can still be obtained with confidence by giving low cost estimates to the conjunct ordering algorithm, namely those found by assuming shortest possible lengths for the clauses, which can be done by ignoring the existence of all literals except those in the rule whose input literals are being ordered. This will usually produce poor lower bounds for the cost of even a complete strategy, so that an additional search would be required to find the optimal sets of directions and the true cost of such a strategy.

Even with exact costs for complete strategies, some pruning of the search space is also useful. For this we will use a "strawman" strategy, one that we believe is worse than the optimal strategy, but whose cost is known. Such a strawman can be obtained by doing a few iterations of local improvement as suggested above, or perhaps in other ways. Certainly any partial strategy whose lower bound estimate is above the estimated cost of the strawman should be discarded. Better yet, any rule whose forwards cost (plus, in a coherent strategy, the forwards costs of its predecessors and so on) exceeds the cost of the strawman can immediately be fixed as a backwards rule, before even generating any partial strategies. Logic programmers often write rules that would, if used forwards, have very large or even infinite numbers of answers, so this is likely to be useful in practice.

Chapter 6

The Incoherent Case

In this chapter, we analyse the results of removing the coherence constraint. This should in most cases allow a strategy of lower cost to be found, because there will no longer be any need to choose a non-optimal direction for some rule in order to be able to use the best direction for one of its predecessors or successors. Indeed a majority of non-toy logic programs written in languages that permit incoherence do in fact make use of it. But the cost of *finding* an optimal incoherent strategy will be larger, because there are more of them than there were coherent strategies (and *a fortiori* the number is exponentially large compared with the number of rules). In fact, the optimisation problem is in general NP-complete, so we give a search algorithm for it. We also discuss algorithms for finding sub-optimal strategies, and what happens when literal ordering is allowed to change.

The NP-completeness proof of Section 4.3, for the coherent case with duplicate answers, applies to the incoherent case if duplicates are possible. However, even if there are no rules which generate duplicates, finding the optimal incoherent strategy can be shown to be NP-complete. The proof of this is very similar to the earlier one, so we will omit some of the framework and concentrate on the differences in details.

6.1 The NP-Completeness Proof

For each vertex v of the graph, create a predicate $P_v(x, y)$ of two arguments, and for each arc from v to another vertex w, create a predicate $P_{vw}(x, y, z)$. The idea is to arrange things so that to deduce P_{vw} we need to deduce at least one of P_v and P_w forwards, and if either of P_v or P_w has been deduced forwards, P_{vw} can be deduced somehow at low additional cost. The difference from the proof of Section 4.3 is that we may choose to deduce P_{vu} forwards.

As before, the proof will apply only to graphs where the number of edges E is large enough for terms of low degree in E to be swamped by the dominant terms.

Define P_{vw} and P_v by the rules

$$Q_{vw}(x)\&P_v(x, y)\&P_w(y, z) \Rightarrow P_{vw}(x, y, z)$$

$$P_{v1}(x, y)\&P_{v2}(x, y) \Rightarrow P_v(x, y)$$

which clearly do not generate any duplicate answers. The arguments x, y, and z all range over the same domain of values, having size d. Q_{vw} is a predicate true of only a few values of x, say n_1 of them, randomly scattered. P_{v1} is true of n_2 pairs of (x, y) values, and P_{v2} of n_3 pairs. The equal frequency assumption is used.

The goal set G will include, for each P_{vw}, n_4 goals of the form $P_{vw}(x, y, K)$, where K is a bound variable.

This gives us a total of $n_2 n_3$ potential resolutions to prove $P_v(x, y)$, each of which succeeds only if both the x and y values match. This has probability d^{-2} of occurring, for an expected $n_2 n_3/d^2$ pairs of (x, y) values such that $P_v(x, y)$ is true. To generate all of these by forwards inference will cost

$I_2 + n_2 L_2$	to look up answers to P_{v1}
$n_2 G_2$	to generate the resolvents
$n_2(I_2 + n_3 L_2/d^2)$	to check if $P_{v2}(x, y)$ is true
$n_2 n_3 G_1/d^2$	to generate the answers themselves
$n_2 n_3 S_1/d^2$	to store them

for a total of

$$c_1 = I_2 + n_2(L_2 + G_2 + I_2) + \frac{n_2 n_3}{d^2}(L_2 + G_1 + S_1)$$

If instead we are given a value for x and use backwards inference to generate all the bindings for y such that $P_v(x, y)$ holds, there will be $n_2 n_3/d^3$ of them. The cost of finding them, excluding the cost of looking up the rule for P_v, will be

$$c_2 = I_2 + \frac{n_2}{d}(L_2 + G_1 + I_2) + \frac{n_2 n_3}{d^3}(L_2 + G_0)$$

If both x and y are bound, the probability that this instance of $P_v(x, y)$ will be provable is $n_2 n_3/d^4$, and the cost of checking it will be

$$c_3 = I_2 + \frac{n_2}{d^2}(L_2 + G_1 + I_2) + \frac{n_2 n_3}{d^4}(L_2 + G_0)$$

The same numbers apply to P_w, of course.

To deduce P_{vw} forwards, we must first obtain the n_1 answers to Q_{vw}. Each of these gives us a value of x, and for each such value we find $n_2 n_3/d^3$ answers to $P_v(x, y)$, getting, on average, $n_1 n_2 n_3/d^3$ answers for P_v. Each of these gives us a value of y which we must substitute into $P_w(y, z)$ and then find the answers for z. The cost of finding these answers if the rule for P_w were used backwards would be $(n_1 n_2 n_3/d^3)(I_2 + L_2 + c_2)$ or

$$\frac{n_1 n_2 n_3}{d^3}(I_2 + L_2 + I_2) + \frac{n_1 n_2^2 n_3}{d^4}(L_2 + G_1 + I_2) + \frac{n_1 n_2^2 n_3^2}{d^6}(L_2 + G_0)$$

whereas the cost of using the rule forwards and then looking up the same answers would be $c_1 + n_1 n_2 n_3(I_2 + (n_2 n_3/d^3)L_2)/d^3$ or

$$I_2 + n_2(L_2 + G_2 + I_2) + \frac{n_2 n_3}{d^2}(L_2 + G_1 + S_1) + \frac{n_1 n_2 n_3}{d^3} I_2 + \frac{n_1 n_2^2 n_3^2}{d^6} L_2$$

We shall arrange that the dominant terms in these expressions shall be the second term of each expression, namely $(n_1 n_2^2 n_3)(L_2 + G_1 + I_2))/(d^4)$ and $n_2(L_2 + G_2 + I_2)$, so that the backwards cost will be larger (for large enough E) if

$$\frac{n_1 n_2 n_3}{d^4} > \frac{L_2 + G_2 + I_2}{L_2 + G_1 + I_2} \tag{23}$$

Under this condition, if P_{vw} is being deduced by forward inference, it is better to do P_w forwards too.

Meanwhile, if we had been using backward inference for P_{vw}, we would have had n_1 values of x for each of n_4 goals, so that we would have tried $n_1 n_4$ times to find the answers for $P_v(x, y)$ given x. So if additionally

$$n_4 \geq \frac{n_2 n_3}{d^3} \tag{24}$$

we see that if P_{vw} is being deduced backwards, the cost for doing P_v backwards will be greater than that for doing it forwards, so that it will be advantageous to do P_v forwards whenever P_{vw} is being deduced backwards.

It follows that, whichever direction is chosen for the P_{vw} rule, the optimal strategy will deduce at least one of P_v and P_w forwards. This assures us that the optimisation problem corresponds to computing some vertex cover of the original graph.

Now, if P_v only has been deduced forwards, the cost of solving the n_4 goals at P_{vw} is

$$\begin{aligned} c_4 &= n_4(I_2 + L_2 + G_3 + I_1 + L_1 + n_1(G_2 + I_2)) \\ &\quad + n_4\left(\frac{n_1 n_2 n_3}{d^3}(L_2 + G_1 + I_2 + L_2 + c_3) + \frac{n_1 n_2^2 n_3^2}{d^7} G_0\right) \end{aligned}$$

If P_w only were deduced forwards, the cost to do P_{vw} forwards plus the cost of looking up the answers to the goals would be

$$\begin{aligned} c_5 &= I_1 + n_1(L_1 + G_3 + I_2 + L_2 + c_2) + \frac{n_1 n_2 n_3}{d^3}(G_2 + I_2) \\ &\quad + \frac{n_1 n_2^2 n_3^2}{d^6}(L_2 + G_1 + S_1) + n_4(I_2 + \frac{n_1 n_2^2 n_3^2}{d^7} L_2) \end{aligned}$$

and if both P_v and P_w were deduced forwards, P_{vw} would become even cheaper.

As in the proof of Section 4.3, in order for the optimal strategy to correspond to a minimal vertex cover, the maximum possible savings in the cost of computing P_{vw} due to doing both of P_v and P_w forwards instead of just one should be at most c_1/E. These savings are bounded above by $\max(c_4, c_5)$. At this point it is convenient to give magnitudes for the

n_i and d. If we set

$$\begin{aligned} d &= E^3 \\ n_1 &= k_1 E \\ n_2 &= E^6 \\ n_3 &= k_3 E^5 \\ n_4 &= k_3 E^2 \end{aligned}$$

where k_1 and k_3 are constants yet to be determined, then the dominant term in the expression for c_1 will be the term $n_2(L_2 + G_2 + I_2)$, which will be proportional to E^6, while in the above expressions for c_4 and c_5, the dominant terms will be, in each case, the first one on the second line, and will be proportional to E^5. Note that the dominant term in c_3 is independent of E and is $2I_2 + L_2 + G_1$.

The coefficients of the dominant terms in c_4 and c_5 are $k_1k_3^2(3I_2 + 3L_2 + 2G_1)$ and $k_1k_3^2(L_2 + G_1 + S_1)$, both of which are less than $k_1k_3^2(3I_2 + 3L_2 + 2G_1 + S_1)$, while the coefficient of the dominant term in c_1 is $L_2 + G_2 + I_2$, so we shall require that

$$k_1k_3^2 \leq \frac{L_2 + G_2 + I_2}{3I_2 + 3L_2 + 2G_1 + S_1} \tag{25}$$

The constraint given above at (23) now becomes

$$k_1k_3 > \frac{L_2 + G_2 + I_2}{L_2 + G_1 + I_2} \tag{26}$$

so set

$$\begin{aligned} k_1 &= 1.08\frac{(3I_2 + 3L_2 + 2G_1 + S_1)(L_2 + G_2 + I_2)}{(L_2 + G_1 + I_2)^2} \\ k_3 &= 0.95\frac{L_2 + G_1 + I_2}{3I_2 + 3L_2 + 2G_1 + S_1} \end{aligned}$$

It is easy to see that there will now be some constant such that if E is greater than this value, all the constraints are satisfied by the resulting values for d and the n_i.

6.2 Lower Bounds on Costs

As was noted in Section 3.3, the cost of using a rule backwards can vary widely across different incoherent strategies, so that we cannot expect to use any optimisation algorithm which requires a single estimate for this cost. Even the forward cost estimates $e_f(r)$ are now subject to variation, due to the possibility of duplicate answers being generated by r's predecessors, which was precluded by the coherence assumption because all these predecessors had to be used forwards. This means that these estimates, which in the coherent case were always the same across all strategies, may have to be re-done each time a new partial strategy is considered.

In general an exhaustive search will be required to find the optimal incoherent strategy. As before, this will be an A^* search in a space of partial strategies, guided by estimates of the least possible cost of a completion of the current partial strategy. We will continue to compute the cost of a partial strategy by adding up the minimum possible costs of all the rules, given the directions specified by that partial strategy. But the space of strategies searched will be much larger without the coherence requirement, so that the number of possible completions of a given partial strategy will usually be much larger than in the coherent case. It also becomes much harder to estimate a lower bound on the backwards cost of a rule, precisely because the set of completions over which we must take the lower bound is larger and more varied.

6.2.1 Minimising rule costs

Recall that we define the cost of a rule as the total cost of all clauses whose first literal was obtained by instantiation from a literal of the rule. To minimise the cost of a forwards rule if it gets (or may get) some of its inputs from backwards rules, we must just minimise the numbers of answers given to it by those rules. To do this, we assume that duplicates are eliminated. For a backwards rule, we must also minimise the number of clauses whose first

literal unifies with the rule's output literal, i.e. the clauses that "invoke" the rule, and this is much harder.

The main reason for this is that, as remarked in Section 3.3, the number of invocations of a rule r used backwards depends on the directions in which its successors are used. As we saw in Section 3.4, every path through the rule graph from r to a goal can give rise to a different set of clauses that resolve with r, provided that all the rules along such a path are used backwards. When some rules above r can be used forwards, every path from r that reaches a forwards rule by passing through backward rules only will give rise to a different set of invoking clauses. So to find a lower bound on the invocations of r, it may be necessary to look at all rules that can be reached from r, and find the directions for all of them that, taken together, yield the least cost at r.

Once we have examined some path from r and found the directions of the rules along it that give the lowest contribution to the cost of r, there is no guarantee that, for some predecessor p of r, the same directions will be among those that *minimise* the cost of p. For example, if r is $A(x,y)\&B(y,z) \Rightarrow C(x,z)$ and p has output literal $A(x,y)$, then two clause sets of equal size whose patterns begin with $\neg C(u,v)$ and $\neg C(u,1)$ (assumed to be generated when some sucessor of r is used either forwards or backwards) will lead to the same set of invocations at p, if r is used backwards, but will probably result in different costs at r: the first set will be more expensive because it has a free variable where the other has a constant. Now if we reduce slightly the size of the first set, it will still give higher cost at r but be less expensive at p. In general, it will be necessary to compute the lower bound cost estimate separately for each rule, and different sets of directions will be found. This has two consequences: the effort of finding a lower bound on the cost of any completion of a partial strategy is increased, and its accuracy is decreased because not all the lower bounds for individual rules can be attained at once.

This would still not be outrageously expensive if there were no instances of reconvergent fanout (see Section 3.4) in the rule graph. With reconvergent fanout, the number of paths

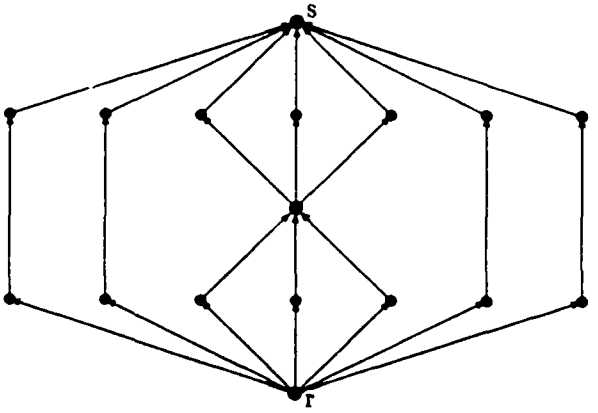


Figure 10: Nested reconvergent fanout

can grow exponentially large compared with the number of rules. To minimise the cost of r we must minimise the size of all the clause sets generated by these paths, which could take exponential time.

It might also happen that different paths going from r to a goal via some rule s would require different directions to be used for s in order to minimise their contributions to the cost of r, so that the apparent minimum cost of r would not be achievable by any consistent assignment of directions to rules. This would force us to choose the better direction for s by adding up the minimal contributions that all these paths would make to the cost of r if s was used backwards, and comparing the result against what would be obtained with s used forwards. There is no guarantee that we could even find these minimal contributions without similar interference occurring at some other rule between r and s, as shown in Figure 10, giving rise to another case split; in the worst case, this could mean that an exponential search would be required merely to find a lower bound on the cost of r. Rather than do this, we may choose to ignore the interactions between paths, and accept a lower bound on r's cost that will probably be unrealistic. This still does nothing to reduce the number of paths that must be examined.

Thus we are stuck with unrealistic lower bounds that take a long time to work out. Mercifully, it is not hard to compute the number of paths from each rule to a goal, so we can tell in advance how long it will take to find the lower bounds. If this turns out to be unacceptably long, we will want a method for computing unrealistic lower bounds without explicitly considering all possible paths.

6.2.2 Coarser lower bounds

The way to achieve this is to generate, at each rule r, clause sets that invoke its predecessors in ways that can be relied upon to yield lower bound cost estimates along every path that leads from some rule to r. These clause sets may be different from any that could actually be generated by assuming either direction for r. We shall show how such sets can be constructed, assuming that corresponding sets for r's successors are available. This construction need only be applied to clause sets that could occur during backwards inference, since we already have a way of finding lower bounds for the cost of using a rule forwards, as mentioned above.

The trick described in Section 3.4.3, of separating clause patterns into heads and tails, with the head consisting of the literals that come from the rule most recently resolved against, offers a reduction in the number of clause sets considered when reconvergent fanout is present, and does no harm when it is absent. The price paid for this can be that sets with different ancestry (different tails) get merged, leading to loss of accuracy, as discussed in Section 3.4.3. To estimate lower bounds on the sizes of sets, we must maintain with each set some information about how the set's pattern and size would change under different completions of the current partial strategy.

There will frequently be some tension between the desire for a clause set itself to be as cheap as possible (which usually means that its size should be minimal) and the desire to minimise the costs of sets obtained from it. For example, by using a rule forwards, we will get some sets that are very small but have patterns with most of their

variables free, and the sets obtained after further resolutions will be larger than might have been obtained from a set with a bigger set size but most of its variables bound. Thus it is useful to keep track of multiple possibilities for each clause set, corresponding to sets of completions of the current partial strategy which lead to more or fewer variables being bound. This will enable us to find tighter bounds for each set of completions than if we computed a single set representing a lower bound valid across all completions.

Each head associated with some rule r will be represented by a clause pattern c that would be obtained if r were used forwards, and the set size obtained using r forwards will be attached to c. For lower bounds, such set sizes should be computed under the assumption that all duplicates are eliminated, i.e. that r's predecessors are used forwards. The additional information kept with c will consist of a list of triples describing possible clause sets that could be generated if r were used backwards: each triple will be made up of a list of bindings for the variables in c, a set size, and a total length for the tails that would be attached to the head of the set that would be generated by using r backwards. There is no need to represent this head explicitly; it would be obtained from c by deleting its last literal (which is an instance of r's output literal) and then substituting in the bindings from the first component of the triple. This binding list would have to contain not only bindings of variables to constants but also a list of free variables that had to become bound. The third compenent of the triple would only be necessary if resolvents were to be generated by explicit substitution of variable bindings into the entire clause; see the discussion in Section 3.4.

Each triple will describe the cheapest possible set of invocations of r that could occur within some subset of the possible completions of the current partial strategy. These subsets will be disjoint and will exhaust the set of completions. The computational speed of the lower bound estimation will depend on the policy adopted for how many such triples to keep, namely how many sets of possible completions to distinguish. The policy to be studied here

$$F(w)\&C(7,u,v) \Rightarrow G(v)$$

$\neg E(X,5)$ $\times 7$

$$D(v,w)\&C(u,v,w) \Rightarrow E(u,w)$$

$D(V,W)$ $\times 4$

0.5 0.2 0.1

$\neg C(u,V,W)E(u,W)$	4.0
$X/u,5/W$	2.8

$$A(x,y)\&B(y,z) \Rightarrow C(x,3,z)$$

$A(1,Y)$ $\times 3$

$\neg B(Y,z)C(1,3,z)$	3.0
$1/u,3/V,W/z$	2.4
$1/X,3/V,5/z$	0.84

$$Y(x,y)\&Z(x,5) \Rightarrow B(y,5)$$

Figure 11: Some rules and resulting triples

relies on dividing the completions up according to how long a path exists from r through backwards rules to a goal or forwards rule.

The first triple will be obtained by assuming that r's immediate successors are used forwards, so that such a path is just 1 arc long. The second triple assumes that at least one of these rules is used backwards, but for any of them that is so used, its immediate successors must be forwards (unless they are goals), so that the longest path from r now has length 2. Subsequent triples add more layers of rules that may be used backwards. Thus the completions considered in forming the k'th triple are like those that were considered for the $(k-1)$st triples of clause sets generated from r's successors. The set size stored with c itself will be regarded as a zeroth triple, with empty binding list.

6.2.3 Computing the bounds

In Figure 11 are illustrated some rules and clause sets, together with some of the triples

to which they give rise. The unification probabilities for some of the variables are also shown attached to their names.

We will explain how to produce the k'th triple for some clause set with pattern c generated from the forwards version of some rule r. In our example, r is taken to be $A(x,y)$ & $B(y,z) \Rightarrow C(x,3,z)$, and c is $\neg B(Y,z)C(1,3,z)$. The figure shows some of the work of assembling the 2nd triple of c. This triple is constructed largely from the triples of c's successors, which are defined to be those clause sets whose pattern has a first literal unifiable with r's output literal, i.e. one that invokes r.

Set sizes

The set size of this triple is found by adding up the contributions from each successor d of c. The only successor whose triples are shown in the figure is $\neg C(u,V,W)E(u,W)$. Since there need be only one path of length k from r through backward rules, the rule from which a particular d was generated may actually be used forwards, or if not, a completion in the set associated with the k'th triple of c need not contain a path as long as $k-1$ from this d to a forward rule. Thus we must take a minimum set size over triples 0 to $k-1$ of d, in order to get the least possible contribution to the set size of the k'th triple of c. There is a further adjustment needed in some cases, which will be explained below.

Meanwhile, the zeroth triple for this particular d shows a set size of 4, which is just the number of $D(V,W)$ facts. The first triple has a size of 2.8, obtained from the 7 goals of form $\neg E(X,5)$ and the 4 facts, giving 28 possible unifcations, multiplied by the probability of successful unification of 5 with W, which is 0.1. So the minimum is $\min(4, 2.8) = 2.8$.

This minimum represents the size of a set of invocations of r by d. To get the size of the resulting contribution to the clause set described by the k'th triple of c, we must multiply this number of invocations by the sizes of the sets of facts that resolve with those of r's input literals that come before c (in this example, just a 3 for $A(1,Y)$), and then by the

appropriate set of unification probabilities. The set size in the zeroth triple of c already reflects all of these set sizes and some of the unification probabilities, namely those relevant to unifications of arguments in the fact patterns against each other (in the example there are no such unifications, because there is only one such fact pattern); the other unification probabilities are those that come from unifications of the bound variables and constants of the $(k-1)$st triple of d with the constants and bound variables appearing either in r's output literal or in the fact patterns that were resolved with r to create c.

All of these constants and bound variables will in fact appear in the last literal of c, which was obtained by substituting into r's output literal the results of the unifications of r's input literals against these fact patterns. This means that we can find the relevant unification probabilities just by considering the unification of d's first literal (an input literal) with c's last literal (c's last literal is a substitution instance of r's output literal, which would, of course, be the first literal of r if r were used backwards). In this example we have $X, V, 5$ from the first triple of d unifying with $1, 3, z$ from the last triple of c, giving probabilities 0.5, 0.2, 1.0. This gives a contribution of $2.8 \times 3 \times 0.5 \times 0.2 \times 1.0 = 0.84$ to the set size of the second triple of $\neg B(Y, z)C(1, 3, z)$. The quantities obtained in this way must now be summed over all successors d of c to obtain the estimated set size for c's k'th triple.

Strictly speaking, we may take advantage of the fact that at least one path of length k from r to a forward rule or goal does exist in the set of completions being considered, by adding a further quantity to the sum of these minima. This quantity is the minimum value, over all successors d of c, of the amount by which the set size of d's $(k-1)$st triple exceeds the minimal set size over triples 0 to $k-1$ of d. At least one of these $(k-1)$st triples represents a set of clauses that will actually be used to invoke r, giving a set of invocations whose size may be larger than the minimum over the first k triples for this particular d. In the example so far, the minimum was obtained at the $(k-1)$st triple, so we cannot add anything.

Bindings

The binding list of the triple being assembled must represent the substitution which, when applied to r's output literal, would give the result of unifying it against the first literals of all of the clause patterns represented by the $(k-1)$st triples of c's successors. This is appropriate because a more bound (more specific) clause will participate in fewer resolutions than a less bound one, and thus yield lower bounds for sets obtained further down the rule graph. There is an exception for constants, discussed below. Remember that a "substitution", in the context of simulated unification, can include changing a free variable into a bound variable of the same name, as well as replacing a variable with a constant. The bindings made in obtaining c's pattern from the forwards version of r must of course also be unified into this substitution. If any of these unifications fail, then the triple cannot be formed.

Variables not appearing in c's first literal can be ignored for this purpose, since the triple will only be used to obtain lower bounds on the sizes of clause sets generated while solving this literal. Thus unifications of variables not appearing in the first literal are irrelevant. Lower bounds on the size of the set of answers to this literal, and hence on the sizes of any clause sets obtained after the answers have been found, can better be found by looking at the sets of facts that could be deduced forwards by the rules it invokes; indeed these bounds are the best possible for the size of the set of answers.

Should two or more of c's successors have binding lists that contradict each other, i.e. have different constants in the same place, this construction will fail to produce a triple even though plenty of clauses might be generated at run-time. This can be avoided by allowing a triple to be replaced by a list of triples obtained from the various successors, representing disjoint clause sets. The triples on such a list would be complementary to each other – that a completion would be assumed to generate either all of them or none of them. Any further triples whose construction made use of such a list would in general also be lists, and if two of c's successors had such lists among their triples, the corresponding "triple" for c

would have to be a product list, including a triple for each pair of compatible triples from the two lists. It could become computationally very expensive to take care of constants in this way.

Constants cause another problem. A clause containing a constant can actually have higher unification probabilities than if the constant were replaced by a bound variable, because a unification of a constant against the same constant succeeds with probability 1.0, but unifying a bound variable against a constant will give a lower probability. Of course, unifying two different constants gives a lower probability yet, namely zero. But this does invalidate the general statement made above, that a more bound clause will participate in fewer resolutions than a less bound one. So those triples of d that contain a bound variable will represent clause patterns that may give rise to smaller sets than if they had constants at the same places. Note that the way in which the binding lists for triples are constructed ensures that the j'th triple of any pattern will be at least as bound as the $(j-1)$st triple.

The k'th triple of c will include in its binding list all the constants which the $(k-1)$st triple of d does, and so there is a danger that the set it represents could give rise to bigger resolvent sets than would arise from the set represented by some lower triple of d having bound variables in place of some of these constants. This can be avoided by means of some extra work during the computation of set size for a triple: to each constant that appears in the k'th triple of c but is absent from some of the triples of some d, must be attached the minimum total set size attainable using only those triples that did not contain the constant. This minimum will in general be higher than the minimum taken over all of the triples.

Then, when the probability is being calculated for a literal represented by this triple to unify with another literal that contains the same constant, we can distinguish two cases. Either the literal represented by the triple will contain the constant, in which case we use the smaller set size for it and 1.0 as the unification probability, or it will contain a bound variable instead, in which case we use the larger set size and the unification probability appropriate

to such a bound variable (obtained by the equal frequency assumption or otherwise). The lesser of the two numbers so obtained is a lower bound on the result. This shows that there is no need to retain the "larger" set size when it is equal to the smaller, or when it is larger than the smaller one divided by the unification probability.

In the example this would work as follows: the constant 5 appears in the first triple of d but not the zeroth. The minimum set size over only the zeroth triple is 4.0, and the unification probability attributable to the 5 unifying against W is 0.1. So the second triple of c must have this 4.0 attached to the constant 5. and in generating third triples for clauses derived from the rule $Y(x,y)\&Z(x,5) \Rightarrow B(y,5)$, we would take the set size for $\neg B(Y,z)C(1,3,z)$ to be $4.0 \times 3 \times 0.5 \times 0.2 \times 1.0 = 1.2$ which when multiplied by 0.1 gives 0.12. This is lower than 0.84 and so must be used instead. What this reflects is that the set of completions described by the second triple of c includes some in which $D(v,w)\&C(u,v,w) \Rightarrow E(u,w)$ is used forwards and $A(x,y)\&B(y,z) \Rightarrow C(x,3,z)$ backwards, so that we would expect to get $4 \times 0.2 \times 3 = 2.4$ clauses like $\neg B(Y,W)C(1,3,W)$, each of which would have a 0.1 chance of resolving against $Y(x,y)\&Z(x,5) \Rightarrow B(y,5)$ to give an expected set size of 0.24. Some information has clearly been lost here, since the lower bound estimate is 0.12; the missing factor of 2 is due to the fact that the difference between the zeroth and first triples of d included a binding of X to u, which has been forgotten in this construction.

This extra work is only done because of the need to obtain lower bounds on the backwards inference that may take place in solving the first literal of c. Thus we do not need to attach extra numbers to constants bound to variables that do not appear in c's first literal, since unifications involving these variables will not occur until after the answers to that literal have been obtained. There is also no need to do this to a constant that is known not to appear in any rule or fact that could be used in an attempt to deduce answers to c's first literal. Thus the 7 in $F(w)\&C(7,u,v) \Rightarrow G(v)$ need prompt no special action.

Tail lengths

The total of tail lengths to be recorded in the k'th triple for c will include the sum over all c's successors of the product of their set size and (head length - 1), since all these heads will definitely resolve against r. But since there may be as few as one successor of r used backwards, the only other contribution to this total is the minimum total tail length found in all of the $(k-1)$st triples for c's successors. If a different policy had been chosen whereby the sets of completions considered were formed on the basis of the *minimum* length of paths throgh the rule graph from r through backward rules toa goal or forward rule, then much stronger lower bounds could be placed on the total length of the tails of clauses associated with c. But since the model of deduction under which tail lengths are important is felt to be impractical anyway, as indicated in Chapter 3, this policy did not seem to have any extra merit.

Known directions

Throughout this construction we have implicitly assumed that all the rules in question are used backwards, or have no direction specified, in the current partial strategy. If a rule is used forwards, then the only triple that exists for any clause pattern obtained from it is the zeroth triple, so parts of this construction will become trivial. Also, if enough rules are used forwards, this may make differently numbered triples of the same clause pattern identical in some cases; we can then discard the higher numbered of such triples. Where a rule is known to be used backwards, patterns generated from it need have no zeroth triples, but can start with the first or higher triple; this might be implemented by means of a dummy zeroth triple containing only a pattern, the higher triples being as described above. A rule all of whose successors were known to be backwards might have no first triple, and so on; some of the minima could be taken over smaller sets of triples if enough rules were known to be backwards.

The lower bound cost of a clause set is then obtained by taking the minimum of the

costs for the sets described by its triples. Summing over all clause sets, we obtain a lower bound on the total estimated cost for completions of the current partial strategy.

6.3 Search Algorithms for the Incoherent Problem

Now that we have these lower bound estimates, we can use them to guide an A^* search of partial strategies, as in Section 4.4. It seems good to grow the partial strategy by working from the top of the rule graph downwards, since this way the rules that are added early will be the ones that have the most effect on the costs of other rules. Uncertainty about the directions of these "influential" rules would cause large inaccuracies in the lower bound estimates, and so would have much potential for directing the search into blind alleys. A slightly better heuristic for the amount of influence a rule exercises over the total cost would be the sum of the most recent lower bound cost estimates for itself and all rules from which paths to it can be found that do not include a forwards rule. This could be approximated at low computational cost by simply ignoring reconvergent fanout and recursively defining the influence of a rule to be the sum of its lower bound cost estimate and the influences of its predecessors. Since this heuristic governs only the order in which rules are added to partial strategies, it affects the speed of the A^* search but not its admissibility, and so the accuracy with which it is calculated is not of paramount importance.

There may be sets of rules such that, despite our best efforts to improve the lower bound cost estimates, the search for an optimal strategy takes longer than we can afford, or are willing to wait. But this should not lead us to abandon the attempt to optimise such a program: there are algorithms which can obtain nearly optimal solutions in shorter time than is taken by A^* to obtain the optimal one. We proceed to discuss a few of these.

The "dynamic weighting" algorithm proposed by Pohl [Poh73] attempts to focus a search in the presence of loose lower bounds by modifying the merit function for search nodes: the estimated cost for getting from the current node to a solution node is multiplied by a factor

which decreases towards 1 as the distance of the current node from a solution node decreases to zero. This is intended to compensate for the under-estimation of this cost by the lower bound procedure. Obviously it can only be done when we know the depth in the search space at which solution nodes appear. In this problem we do know it, since it is just the number of rules.

The rationale for changing the merit function like this is that search nodes whose cost is mostly known will be preferred over nodes whose cost is about the same but is not well known; admissibility is not sacrificed, since the distortion of the merit function is eliminated as we approach the solution. Unfortunately, it is hard to see how large the factor used to weight the merit function should be, and how it should vary across different rule graphs. Thus Pohl's algorithm seems to be of limited use, and the cost of searching the space of partial strategies may still be unacceptably high in bad cases.

Algorithms have been developed which modify A^* search to find a solution within some factor of the optimal cost. Both the A_ϵ^* algorithm reported in [PK82] and the A_ϵ search of [GA83] allow the user to specify the ratio by which the cost of the solution found may exceed the optimal cost, though the reduction in search effort of these algorithms over standard A^* has not been analysed.

One could iteratively carry out an A_ϵ^* or A_ϵ search with decreasing values of ϵ until the improvement in the solution obtained was roughly equal in value to the effort of the last search. It seems reasonable to assume that a law of diminishing returns will apply as we ask for solutions closer and closer to optimal, in other words the marginal improvement in the cost of the strategy obtained will become smaller as a proportion of the extra effort expended on search. The exponential behaviour of A^* search for all but the most accurate heuristic functions [Pea83] leads us to suspect that the last search might be considerably more expensive than the next-to-last unless ϵ is reduced in very cautious steps.

The "iterative deepening" search technique described in [Kor85], and somewhat differently in [ST85], can also be used to find near-optimal solutions at some potential saving in

time. Briefly, this technique involves conducting an exhaustive depth-first search bounded by some threshold depth, and if no solution is found, increasing the threshold slightly and repeating; its main claimed advantage over A^* search is that it requires far less storage for search nodes, while finding the optimal solution almost as fast. It could be modified to find near-optimal solutions quicker than A^* by using a larger increment in the threshold than would be dictated by Korf's criterion, and returning the first solution node found. The cited authors have analysed the running time of this search algorithm relative to A^* search. They conclude that the actual saving of time, if any, would be determined by the *effective branching factor*, which is the increase in number of nodes in the search space caused by a given increase in the threshold. It is not clear how to predict this factor; it would have to be found empirically while the search was going on.

Either way, we have a problem of controlling the control computation, in the sense of deciding how much of a particular kind of optimisation is profitable to do. Some of the pitfalls inherent in doing this will be discussed in Section 8.2. Since we do expect to be able to compute upper bound estimates for partial strategies (the method for finding lower bounds on clause sets can obviously be reversed to find upper bounds as well), the suggestion made at the end of Chapter 4 applies here also, namely the search could be stopped as soon as a strategy with an acceptable upper bound was found.

6.4 Choosing Directions Without Search

The availability of lower and upper bound estimates for costs opens up the tempting prospect of computing these bounds for each rule in each direction, and if there are any rules where the upper bound for one direction is less than the lower bound for the other, we can decide their directions without more ado. Even though we can hardly expect to find the optimal strategy this way, it could be used to eliminate many possible partial strategies before embarking on a full search. Once a search has been started, we may be able to

determine that under the choices specified by a partial strategy, some rules now have one direction that is definitely better; these can be added immediately to the partial strategy, thus reducing the number of completions of it that must be considered.

Unfortunately, the correct direction for a rule r is not necessarily the one giving the lower estimated cost. Changing the direction of one rule can cause changes to the costs of other rules which outweigh any savings made on the first rule. Consider the rules

$$\begin{aligned} Q_{vw}(x) \,\&\, P_v(x,y) \,\&\, P_w(y,z) &\Rightarrow P_{vw}(x,y,z) \\ P_{v1}(x,y) \,\&\, P_{v2}(x,y) &\Rightarrow P_v(x,y) \\ P_{w1}(x,y) \,\&\, P_{w2}(x,y) &\Rightarrow P_w(x,y) \end{aligned}$$

from Section 6.1. Assume that the rules for P_v and P_w are being used backwards. Then if the rule for P_{vw} is used backwards, the cost of the P_v rule will be $n_1 n_4 c_2$ and that of the P_w rule will be $n_1 n_2 n_3 n_4 c_3 / d^3$, whereas if the P_{vw} rule is used forwards, these numbers change to $n_1 c_2$ and $n_1 n_2^2 n_3^2 c_2 / d^6$. With the magnitudes used in Section 6.1, these numbers are proportional to E^6, E^5, E^4, and E^6. Since the cost of the P_{vw} rule was in no case larger than some constant times E^5, it is clear that the dominant influence on the total cost will be the constant factors in the E^6 terms, and the costs for the P_{vw} rule will probably not determine its direction in the optimal strategy.

It would appear, then, that we cannot be sure of choosing the right direction for a rule r without knowing what the effects of changing r's direction will be on the rules below it. The correct way to find this out is probably to continue with the search and explicitly generate partial strategies using both directions for r. This can be done quite cheaply as long as the cost estimation code uses some form of truth maintenance, so that numbers from the previous partial strategy that are not affected by r's direction can be carried over unchanged. A "strawman" strategy, as described in Chapter 5, could also be used to rule out the forwards direction for some rules, and possibly to inhibit further computation of cost estimates in a partial strategy if the total of the lower bound values exceeded the strawman

cost before all the rules had been processed.

6.5 Re-ordering Input Literals

The negative results of Chapter 5 still apply to the incoherent case, but the result about estimating the costs of forwards rules does not, since it relies on coherence. We can, of course, still use the results if we have a contiguous group of rules known to be used forwards, or backwards, but such groups will generally be smaller than in the coherent case. The methods for estimating lower bounds on expenses still apply to any rule having at least some predecessors used backwards (and if the predecessors are all forwards, then we are in luck). But in the early stages of a search, with most directions unknown, there is little help available.

We can still use "strawman"-based pruning, and can still augment partial strategies by adding first the expensive predecessors of rules in the strategy, then eventually cheap ones. But there is no need to start at the goals G and work downwards; any rule can be added to the strategy at any time. Of course, if the rule is added as a backwards rule and is not connected to a goal or forwards rule by a chain of backwards rules of the strategy, we will have no information about the goals that it will solve, and it will make no difference to the lower bound estimates. Still, any rule when added as a forwards rule will immediately increase the lower bound by an amount equal to the result of running the modified Smith algorithm on it, assuming that all its inputs are looked up. Rules with high costs obtainable this way should probably be added early, with the aim of proving that they must be used backwards. It remains generally true that high-cost rules should be brought in to a partial strategy early, since this will cause very bad strategies to be detected early, and will reduce the inaccuracy in the estimated costs of good partial strategies.

Chapter 7

The Hybrid Case

In this chapter we abandon the restriction that the output literal of a rule has to be at one end of it. When the order of the input literals is fixed, finding the optimal strategy is not much harder than in the incoherent case. If we allow the input literals to be permuted, the optimisation problem remains hard but changes its character somewhat.

7.1 Cost Estimation

The presumed meaning of a strategy (described by giving the position of the output literal in each rule and the order of its input literals) is a little more complex than in the incoherent case. Each rule that has any input literal appearing before its output literal is initially run as a forwards rule, and stops when resolvents are obtained that begin with substitution instances of the output literal; these are stored in the database and subsequently used as backwards rules when a goal is being worked on. Thus the forwards part of the deduction in this case can be thought of as generating new backward rules by "folding" known facts into rule schemata. In what follows, these new rules will be called the "generated" rules.

The modelling of deduction cost does not change much from the incoherent case. The cost of looking up backwards rules must now be estimated differently, because the set of

rules that will be in the database is no longer known, but contains some generated rules, whose number must be estimated.

7.2 Fixed Input Literal Order

7.2.1 Relationship to the incoherent problem

The number of possible hybrid strategies is larger than the number of incoherent ones for a given rule set, since the number of positions for the output literal in a rule with n input literals is $n+1$ instead of 2. So the problem seems harder than the incoherent one, but not by very much (if input literal ordering were also considered, the comparable number would be $(n+1)!$). An NP-completeness proof for it could probably be based on the one for the incoherent case, by designing rules for which the only placements of the output literal that made any sense were at either end of the rule; this could be done by placing some literals at the beginning of the rule that were very expensive to use forwards, and some at the end that were expensive to use backwards, so that placing the output literal in the middle would incur high costs both ways, and thus would never be optimal.

The relationship of the hybrid case to the incoherent case is best appreciated in the light of the following construction:

Given a rule $A_1 \,\&\, A_2 \,\&\, \ldots \,\&\, A_n \Rightarrow B$

by adding suitable new literals B_i for i from 1 to $n-1$ we can decompose it into the rules

$$\begin{aligned} A_1 &\Rightarrow B_1 \\ B_1 \,\&\, A_2 &\Rightarrow B_2 \\ B_2 \,\&\, A_3 &\Rightarrow B_3 \\ &\ldots \\ B_{n-1} \,\&\, A_n &\Rightarrow B \end{aligned}$$

so that each B_i stands in for the new rule obtained by resolving away the first i literals of

the original rule. By using the first j of these "decomposed" rules forwards, and the rest backwards, we would do very much the same deductions as if we had used the original rule with the output literal B positioned between A_j and A_{j+1}. Therefore, this rule decomposition gives a mapping from hybrid strategies on the original rule set to incoherent ones on the decomposed rules, with coherence constraints applying only between the directions of rules derived from the same original rule.

The cost of a hybrid strategy would not be quite the same as that of the incoherent strategy produced from it by this decomposition. There would obviously be extra storage costs for the decomposed rules and the B_i, and some extra processing cost for looking up the B_i when using the decomposed rules.

7.2.2 Lower Bounds

The rule decomposition also makes it clear that the set of subgoals beginning with $\neg A_j$ depends only on the direction of the j'th decomposed rule and the set of answers found for B_{j-1}. Translating back into the original formulation of the problem, these become the position of A_j relative to the output literal B, and the set of answers obtained to the conjunction of the literals before A_j.

If B comes after A_j, then only forward inference will be involved in generating clauses beginning with $\neg A_j$, so as far as A_j is concerned this looks just like an incoherent strategy in which the whole rule is used forwards; the same subgoal set will be generated. If B comes before A_j, then for any goal clause beginning with $\neg B$ and any one list of input facts unifying with the the A_i, the same clause beginning with $\neg A_j$ will be generated independently of where B appears before A_j. However, different lists of input facts might give rise to the same clause, in other words duplication is possible, and will be affected by the position of B. Say it comes between A_k and A_{k+1}. Since forward inference will be used by the first k decomposed rules, if there is more than one fact unifying with some of the A_i for $i < k$, some duplicates may be eliminated in these rules. Translating back to

the hybrid formulation, this means that clauses beginning with $B\neg A_{k+1}\ldots$ will be stored and duplicates of such clauses will be eliminated. Either way, it is clear that the number of clauses beginning with $\neg A_j$ may be lower than if pure backwards inference had been used.

We see, therefore, that the methods of Chapter 6 for estimating clause sets and finding lower bounds on costs can be carried over almost unchanged, once allowance has been made for this elimination of duplicates.

To estimate the minimal cost of using a rule, once a suitable set of subgoals has been found, still requires that a position for the output literal be determined, which was not necessary in the incoherent case. But this can be done by just assuming that all duplicate answers to the A_i are eliminated, and then enumerating all the possible positions for the output literal, which will not take long (the alternative, namely formulating the placement of the output literal as a coherent problem over the set of new rules and solving it, would probably take longer). The number of new rules that have to be stored may turn out to be a major influence on the cost, which makes this slightly different from the coheent case where only units had to be stored (this is presumably cheaper than storing longer clauses).

7.2.3 Searching the space

Unless pruning proves very effective indeed, the search space will still be larger than in the incoherent case with fixed input literal order. But a fairly convincing lower bound estimate of the cost of a partial strategy is available. This lower bound is in fact slightly more convincing than before, since it is now sometimes legitimate to assume that different literals from the same rule will be used in different directions; this implicit assumption was a source of potential inaccuracy in the lower bound estimates of Chapter 6. The same search algorithms as were recommended for the incoherent case should also work fairly well here.

7.3 Reordering Input Literals

If we both allow the output literal to move around and the order of the input literals to vary, this amounts to arbitrary permutation of all the literals of each rule. The resulting search space is not dramatically larger than that obtained when only input literals are permuted, but slightly different lower bound estimation methods are needed.

The decomposition described above suggests that what we should do in searching for the optimal strategy is augment partial strategies by adding new directions for one literal at a time. This amounts to a decision whether to put a particular input literal before or after the output literal. But to estimate the lower bounds, it may be necessary to assume positions for every input literal. If an input literal is before the output literal, it will contribute directly to the cost of using the forwards part of the rule, and will indirectly affect the expense of solving subgoals, via its effect on the set of backwards rules generated. If it is after it, it can only affect the expense of solving subgoals.

7.3.1 Lower bounds on forward inference cost

Referring to Section 5.4.3, it is clear that if we ignore the expenses of the "low-volume" literals of a rule, then they can come before all other literals, since this will reduce the cost of the forward literals directly and also cause fewer new backwards rules to be generated from the original rule. So for purposes of finding lower bounds on the cost of the forward part of a rule, it is not sufficient to then take the literals known to be used forwards and order just these. We must include the low-volume literals, and there may be other literals that become low-volume after variables appearing in one of the forward literals have been bound; such literals should then be added to the sequence of literals being grown by the ordering algorithm.

The correct way to obtain a lower bound for the forwards cost is to run a suitably modified version of the Smith algorithm on those literals of the rule that are *not* known

to be *backwards*, with a different termination criterion than before: the search terminates once the output literal of the rule has been added, even if there are still some other literals left over. There is, though, a constraint that this cannot be done until all of the forwards literals have appeared. Note that this forces inclusion of the cost of generating and storing the new rules.

7.3.2 Lower bounds on backward inference cost

For lower bounds on the expense of the backwards part of a rule when invoked by some subgoal, it would be absurd to generate all possibilities for the forwards part of the rule and then find, for each of them, the best ordering of the remaining literals. Instead, we can run our literal ordering algorithm in reverse, generating final subsequences rather than initial subsequences, using all literals that are not known to be forwards. As above, the search terminates when the output literal has been added, which must not happen until all the backwards literals have been.

This search is less easy to do, because we cannot directly calculate the cost of a final subsequence; it must be found by knowing the number of answers to the goal and working backwards to the intermediate *Numsol* values, from which we then obtain the cost. In turn, the number of answers is not known exactly, for it depends on the number of duplicates eliminated at the end of the forwards part of the rule. This can be dealt with by assuming initially that all duplicates are eliminated, which will certainly give a lower bound; if the proportion of duplicates is large enough to be of concern, we can re-calculate it each time the final segment grows longer (this reduces the set of literals that could have been used forwards, and hence the amount of duplication). The variables that will be bound when a literal is resolved upon are just those that appear in literals that have not yet been placed, including the forward literals.

This search still benefits from the Adjacency Restriction (suitably modified) and from the results about permutations of sequences of literals. The result about low-volume literals

does not apply, since it can only force us to place them at the beginning of the ordering, and this search does not know where the beginning is. Unfortunately, no comparable result about placing high-volume literals at the end is true. The "cheapest-first" and "tree query" algorithms mentioned in Section 5.2 do not seem to be applicable when the set of literals involved, as well as their order, is variable.

The upshot of all this is that finding the optimal hybrid strategy with arbitrary permutation of literals can be expected to take substantially longer than in the incoherent case, mainly because there are more directions to be chosen. It is also going to be hard to find a good "strawman" strategy whereby the search can be pruned, and harder to do the pruning: the cost occasioned by using a single literal forwards is much less likely to exceed the cost of a strawman than is the cost occasioned by using a whole rule forwards, so there will be fewer directions that can be fixed in advance.

7.4 Random Algorithms

As compensation for these troubles, a more powerful tool for local improvement becomes available. Since all literals of the rule are now being permuted, there is no longer any need to optimise input literal order separately from output literal placement. We can just use a suitable literal ordering algorithm to find the least-cost permutation of all the literals. Of course, the output literal does not have an expense in the same way as input literals do; the computational cost attributable to a decision to put the output literal in a particular place is just the cost of storing the clauses beginning with that literal.

An optimisation algorithm based solely on iterated local improvement now suffers more than ever from positive feedback, which applies both to output literal placement and to input literal permutation. So we may expect it to head even more quickly for a local optimum. A partial remedy for this would be to run it several times, randomising the order in which rules are improved: since the change made to one rule usually depends on what

has been done to other rules, this could cause the algorithm to move in different directions from a given starting point, and so find several different local optima. It does not, however, guarantee anything.

Methods do exist for obtaining good, though not necessarily optimal, solutions to problems that are combinatorially explosive with no good heuristics known. They depend on generating a set of random solutions and improving it by stochastic transformations under the guidance of some merit function. Two of the best known are "genetic" or "adaptive" search and "simulated annealing" or "statistical cooling".

7.4.1 Genetic Search

Genetic search was originally devised by Holland [Hol62,Hol75], with subsequent work reported in many places, for example [Mau84]. It works by creating a "population" of random solutions and allowing it to evolve. Evolution is driven by sexual reproduction and by mutation. Each strategy is represented as a string of numbers or bits, all strings being the same length, say N. Two strategies may add children to the population, in a number determined by the combination of their own merit values; each child is generated by appending the last n elements of one parent string to the first $N - n$ of the other, where n is randomly chosen for each child. A strategy can mutate by randomly changing one of the elements of its string. There is a mechanism for old strategies to die off. Given a large enough initial population, and enough generations of evolution, something good should result.

It is, however, difficult to tell how large the population should be and how long it should be allowed to evolve for. Moreover, in order to ease the labour of estimating the costs of newly generated strategies, it is helpful to keep the data structures that were generated while estimating the costs of the previous strategies; with a large population, this could become costly. There are some minor problems with representing an arbitrary strategy in suitable form, so that the above-mentioned string transformations would yield strategies bearing a sensible relationship to their parents.

7.4.2 Simulated Annealing

Simulated annealing [KGV82] keeps only one strategy and continually tries random perturbations to it. If the perturbed strategy is better, it is kept and the old one thrown away. If it is worse by an amount δ, it is kept with probability $e^{-\delta/T}$ where T is a parameter called the temperature, otherwise the old strategy is kept. This makes it possible for the algorithm to travel from a local optimum to another optimum which is separated from it by a "ridge" of non-optimal solutions. By gradually reducing T over a large number of iterations, we have a good chance of obtaining a near-optimal strategy. The method derives its name from the analogy with the process of annealing a hot piece of metal by cooling it gradually, so as to obtain a crystal structure of lowest energy.

An advantage of simulated annealing is that a quantity analogous to entropy can be defined, and can be estimated as the algorithm proceeds. Using thermodynamic theory, the entropy value yields an indication of how far the current solution is from optimal, and whether the temperature is being decreased too quickly. This is discussed in detail in [AvL85].

7.4.3 Random starting points

In [AvL85] it is also observed that for one type of problem studied (a version of the Travelling Salesman Problem), simulated annealing had performance equal to a much simpler method. The simpler method was to start from random strategies and use repeated local improvement until a local optimum was reached. After running this for 5000 seconds, the best strategy found was little worse than that obtained by simulated annealing in the same length of time.

It is unclear whether this situation would recur in a different optimisation problem, such as ours. The possibility is quite tempting, though, because a single calculation of expense estimates would enable us to do local improvements to every rule's input literal order, thereby yielding a great deal of "bang for the buck" compared with any method

that re-calculated estimates after a single change to the strategy. By contrast, simulated annealing relies on being able to re-evalute the cost of a strategy after a perturbation very quickly, which probably makes it unsuitable for our problem. Experience with simulated annealing [Sua86] shows that it performs best when the cost function can be tailored to the features of the problem at hand, whereas the estimated cost for a strategy is not amenable to this.

All three of the methods discussed here have the advantage that they can be stopped at any time if the optimiser decides it is satisfied with the best strategy obtained so far, and does not wish to expend more effort on looking for a better one. See also the discussion of "controlling control" in Section 8.2.

Chapter 8

Practical Considerations

In this chapter we address two related issues, both having to do with the accuracy of the results returned by the optimiser. The first of these is the possibility of error in the cost estimates used in the choice of a strategy; this could lead to a sub-optimal strategy being returned as "optimal" and to the predicted cost of using it being wrong. The second is the trade-off between the cost of optimisation and the cost of eventual execution of the optimised program: even with perfect cost estimation, it might be better to return a strategy that is not known to be optimal, if its cost is known to be low enough that the effort of further optimisation is unnecessary or unjustifiable.

8.1 Impact of Errors

Throughout this dissertation we have usually referred to "estimated costs" because we do not expect to be able to predict the cost of non-trivial deductive programs with absolute accuracy. In this section we discuss the sources of error in such estimation and the extent to which the "optimised" strategies can be trusted in the presence of errors. We obtain bounds on the amount of resources wasted by using erroneously obtained strategies, and describe some conditions under which erroneously chosen strategies can still be useful.

8.1.1 Errors in the estimates

We assume that the software that will run the logic program is well understood, so that the main source of error in the cost estimates will be errors in the estimated numbers of resolvents. There are three possible sources of such errors. These are

- errors in the input data (set sizes, domain sizes, and unification probabilities)
- departures from the equal frequency assumption (where used)
- random fluctuations within a probability distribution

It is also possible, as mentioned in Section 3.4, that some error could arise as a result of merging clause sets whose patterns had similar heads.

Input data errors

If incorrect numbers are supplied by the user, it is important to know how bad the resulting cost estimates will be. Since most of the operations involved in estimating costs are multiplications and divisions, we will look at the proportional errors in the estimates as a function of the proportional errors in the inputs to the simulator.

Assuming for the moment that the errors expected are symmetric about the values supplied, the proportional error of a product or quotient is simply the sum of the proportional errors of its arguments. That of a sum can be found by converting to absolute errors, adding these, and then dividing by the estimate for the result: if we add two quantities, one of which has expected value a with proportional error x and the other b and y, then the result will have expected value $a+b$ and error $(ax+by)/(a+b)$. Another bound on the proportional error of the sum is $\max(x, y)$. There is no simple formula for the error in a combinatorial expression like that in equation 5 from Section 3.1.5, and since the value of the expression there is a non-linear function of the quantities g, h, and m, the error resulting from a given upward error in one of these may differ from that caused by a downward error of the same

size. Probably the best thing to do about this is to re-calculate the entire expression using upper and lower values for the quantities expected to be in error.

It will in any case be better to separate the errors into upwards and downwards components, especially since the upwards error (amount by which the actual value may exceed the estimate, as a proportion of the estimate) may exceed 1, while it would be absurd for the downwards error to do so. To calculate the errors of a quotient we must now swap the proportional errors of its denominator and apply a suitable transformation ($x \rightarrow \frac{1}{1+x}$ or $x \rightarrow \frac{1}{1-x}$) before combining them with the errors of the numerator. Given upwards and downwards error bounds for the input data, we can calculate upper and lower bounds on our estimates. This will be important for the "optimistic" and "pessimistic" cost estimates mentioned in several places. However, it is not immediately obvious which errors are optimistic.

Clearly, the more facts and goals there are, the more resolvents will be generated. Increased unification probabilities, if these are given separately from domain sizes (i.e. the equal frequency assumption is overridden), also cause more resolvents to be generated. The effect of changes in domain sizes depends both on whether the equal frequency assumption is in force and on whether duplicates are being eliminated. If they are not, referring to Section 3.1.4, we see that domain sizes do not appear explicitly in equations 1 and 2, so increases in domain sizes will either have no effect on the numbers of resolvents generated, or, if the equal frequency assumption is in force, will reduce these numbers, via a reduction in unification probabilities.

When duplicates are being eliminated, equation 5 from Section 3.1.5 must be used, and it is intuitively obvious that the number of unique answers will increase when g increases, since there is now room for more answers before some of them have to become duplicates of each other. It is equally clear that the proportional increase in the number of answers can be at most as big as that in g. Moreover, this assumes that the increased domain sizes that caused the increase in g did not lead to any change in m, the number of answers before

elimination of duplicates. But if any of the domain sizes that make up g were used under the equal frequency assumption to estimate the probability of a unification, m would be reduced by increases in these domain sizes, cancelling out the effect of the increase in g, and in fact reversing it. An increase in h means an increase in the number of ways in which each possible unique answer can be duplicated, which means that duplicates can occur more easily and the number of unique answers will be the same or somewhat smaller. So the cost estimates can be both increased and decreased by increases to the domain sizes, depending on the details of each rule that generates duplicates.

Dependencies among the input data

The cost estimation is done by manipulating three sorts of quantities: numbers of resolvents generated, numbers of potential instances of patterns (found by multiplying domain sizes), and probabilities of a typical potential instance of a pattern being generated. The equal frequency assumption, where it is used, gives rise to dependencies between these quantities, by linking unification probabilities to domain sizes. If the equal frequency assumption was used erroneously, the unification probabilities derived therefrom will be in error by different amounts than the domain sizes themselves. There is no way to disentangle these errors while still using the assumption, since the system cannot know whether the number given by the user was meant as a domain size or as the reciprocal of a unification probability; it is conceivable that users would be sure of one of these quantities but not of the other.

Random noise errors

If we have two constant values drawn from a uniform distribution on n values, then the probability of their being equal is $1/n$. If we have m such pairs, the expected number of pairs of equal values is m/n, and the variance of this number is also m/n. The standard deviation is $\sqrt{m/n}$, so as the expected number of successful resolutions increases, the expected proportional error in the number decreases. Thus in general, for programs that will

perform large numbers of resolutions, errors due to this kind of randomness will be small relative to the cost of running the program. For cheap programs, higher proportional errors will arise. This sort of error is not a defect of the simulation, but reflects (presumably) real uncertainty about what will happen when the program runs, so it should be taken into account by any optimiser that wants to reduce the average running time over all likely executions of the program.

8.1.2 Effects of errors

If the costs attached to a rule are in error, then in general the strategy returned will be wrong. It is tempting to observe that errors which increase the cost of forward inference are likely also to increase the cost of backward inference, and so disturb the choice of strategy rather less than might be expected. Such behaviour has been observed in database query optimisation [ASK80], where the correct method for executing a query was picked even when the optimiser had erroneous cost estimates. Unfortunately, it is easy to construct examples where a particular error affects one cost more strongly than the other, so that any general treatment would be unable to rely very much on this "linkage".

It is easy to construct a case in which small errors in the costs could result in large changes to the strategy; however, the actual cost of using the erroneously obtained strategy would be close to the estimated cost, since the errors were small. Intuitively, we can say that small errors in costs will only lead to large changes in strategy if there were two different strategies with almost the same cost, so that the optimiser was finely balanced between them, and it mattered little which one was chosen. It also turns out that the truly optimal cost, which would have been obtained if the correct strategy had been found despite the errors, is close to the estimated cost. In this section we obtain bounds on the differences between these costs, in terms of the errors in the cost estimates for the rules.

Call the cost returned by the optimiser using our cost estimates the *estimated cost* C_E. Define the *actual cost* C_A to be the true cost of executing the strategy found by the

optimiser, and the *best cost* C_B to be the cost of the strategy which the optimiser would have obtained if it had had the true costs as inputs. $C_A - C_E$ and $C_E - C_B$ can both be thought of as measures of how badly the output of the optimiser is misleading the user, and $C_A - C_B$ is the amount of money we are causing to be wasted. All these numbers are of some interest.

Effects on the linear program

In the cases where the linear programming methods of Chapter 4 are used, recalling that the estimated costs are written $e_f(x)$ and $e_b(x)$, let the corresponding true costs be $c_f(x)$ and $c_b(x)$ and the errors be

$$d_f(x) = e_f(x) - c_f(x)$$

$$d_b(x) = e_b(x) - c_b(x)$$

Also define

$$\begin{aligned} dmax(x) &= \max(|d_f(x)|, |d_b(x)|) \\ dsum(x) &= |d_f(x)| + |d_b(x)| \end{aligned}$$

It is easy to see that

$$|C_E - C_A| \leq \sum_x dmax(x)$$

Now recall that the optimal set of rules to use forwards is written R_f, and define R_b to be the difference of R and R_f; define R'_f and R'_b to be the sets obtained by the optimiser. By definition,

$$\begin{aligned} C_E &= \sum_{R'_f} e_f(x) + \sum_{R'_b} e_b(x) \\ C_B &= \sum_{R_f} c_f(x) + \sum_{R_b} c_b(x) \\ C_E - C_B &= \sum_{R_f \cap R'_f} d_f(x) + \sum_{R'_f \setminus R_f} e_f(x) - \sum_{R_f \setminus R'_f} c_f(x) \\ &\quad + \sum_{R_b \cap R'_b} d_b(x) + \sum_{R'_b \setminus R_b} e_b(x) - \sum_{R_b \setminus R'_b} c_b(x) \end{aligned} \tag{27}$$

But $R'_b \setminus R_b = R_f \setminus R'_f$ and $R'_f \setminus R_f = R_b \setminus R'_b$, and by the definitions of these sets of rules,

$$\sum_{R'_b \setminus R_b} e_b(x) \leq \sum_{R_f \setminus R'_f} e_f(x) \tag{28}$$

$$\sum_{R'_f \setminus R_f} e_f(x) \leq \sum_{R_b \setminus R'_b} e_b(x) \tag{29}$$

so by subtraction

$$\sum_{R'_b \setminus R_b} e_b(x) - \sum_{R_f \setminus R'_f} c_f(x) \leq \sum_{R_f \setminus R'_f} d_f(x)$$

$$\sum_{R'_f \setminus R_f} e_f(x) - \sum_{R_b \setminus R'_b} c_b(x) \leq \sum_{R_b \setminus R'_b} d_b(x)$$

and, substituting into (27),

$$\begin{aligned} C_E - C_B &\leq \sum_{R_f \cap R'_f} d_f(x) + \sum_{R_f \setminus R'_f} d_f(x) + \sum_{R_b \cap R'_b} d_b(x) + \sum_{R_b \setminus R'_b} d_b(x) \\ C_E - C_B &\leq \sum_{R_f} d_f(x) + \sum_{R_b} d_b(x) \\ C_E - C_B &\leq \sum_{R} dmax(x) \end{aligned}$$

which is very reassuring.

We also get

$$C_A - C_B = \sum_{R'_f \setminus R_f} c_f(x) - \sum_{R_f \setminus R'_f} c_f(x) + \sum_{R'_b \setminus R_b} c_b(x) - \sum_{R_b \setminus R'_b} c_b(x)$$

Rewriting (28) and (29) as

$$\begin{aligned} \sum_{R'_b \setminus R_b} [c_b(x) - c_f(x)] &\leq \sum_{R_f \setminus R'_f} [d_f(x) - d_b(x)] \\ \sum_{R'_f \setminus R_f} [c_f(x) - c_b(x)] &\leq \sum_{R_b \setminus R'_b} [d_b(x) - d_f(x)] \end{aligned}$$

we obtain

$$C_A - C_B \leq \sum_{R_b \setminus R'_b} [d_b(x) - d_f(x)] + \sum_{R_f \setminus R'_f} [d_f(x) - d_b(x)]$$

so that, if R_1 is the set of rules x such that $d_f(x)$ and $d_b(x)$ are known to have the same sign (for example, it might be known that both e_f and e_b were underestimates) and R_2 the rest of R,

$$C_A - C_B \leq \sum_{R_1} dmax(x) + \sum_{R_2} dsum(x)$$

This is worse than the error in C_E by a factor of at most 2.

These results can be applied as they stand to the choice of optimal coherent strategy with fixed input literal order on a logic program with no rules capable of generating duplicates; if there are duplicates, then we must re-define $d_b(x)$, since it can now depend on the directions of rules other than x. For the purposes of finding a bound on $|C_E - C_A|$, we can take $d_b(x)$ to be the worst error obtainable given the directions chosen by the optimiser. When comparing C_B with either C_E or C_A, we should take $c_b(x)$ to be the cost actually incurred using the actually optimal strategy, and $e_b(x)$ to be the estimated cost using the optimiser's strategy. It is, of course, impossible to compute $c_b(x)$, since we do not know the optimal strategy. However, if we know the maximum proportional error that can occur in $e_b(x)$, we can compute a value for $e_b(x)$ under the assumption that all duplicates disappear, and then apply the proportional error factor to get a lower bound on $c_b(x)$. An upper bound can be computed similarly. This gives us upper and lower bounds on $d_b(x)$ and on $|d_b(x)|$.

Effects on the search algorithms

In the incoherent or chaotic cases, or when literal order is changeable, the foregoing analysis is invalid, because we will not have well-defined values for the absolute errors. We can, however, still discuss the proportional errors in C_A and C_E.

If the ratio of estimated cost to true cost is known to be between k and K for every rule, namely

$$k \leq \frac{e_i(x)}{c_i(x)} \leq K$$

where $k \leq 1 \leq K$, then it is clear that

$$kC_A \leq C_E \leq KC_A.$$

Moreover, by the definition of C_B, $C_B \leq C_A$, so $C_B \leq C_E/k$. To obtain a lower bound for C_B in terms of C_E, observe that if the optimisation algorithm were given estimated costs that were below the true costs for every rule, it would return a C_E that was below C_B. But if we multiply the estimated costs by $1/K$ then they will satisfy this, and the optimiser would return C_E/K if we gave it these costs. Thus $C_E/K \leq C_B$, and

$$kC_B \leq C_E \leq KC_B.$$

We now obtain

$$C_B \leq C_A \leq (K/k)C_B$$

and this lower bound for C_B in terms of C_A is in a sense the best possible, as can be seen by considering the case where the estimated costs for forward and backward inference are equal, but the true costs for backward inference are $1/k$ times the estimates, and for forward inference are $1/K$ times the estimates. If the optimisation algorithm arbitrarily decides to infer everything backwards, we will get $C_A = C_E/k$ whereas clearly $C_B = C_E/K$.

This approach is not very satisfactory, because the bounds k and K may be attained only at sets of clauses which do not appear in any strategy that would be returned by the optimiser, either using the estimated or the true costs. To find the maximum plausible error in the size of any clause set that might actually be generated in an optimal strategy is going to be very difficult. To obtain bounds on C_A is relatively easy, since we need only re-do the estimation of total cost for the strategy already chosen. But to find bounds for C_B we must repeat the entire optimisation process, once using optimistic values of the input data and once pessimistically.

8.1.3 Use of erroneous strategies

The optimisation algorithms presented in earlier chapters do not take any stand on whether they are choosing strategies for the best case, average case, worst case, or any other case of the input data. One can easily imagine a user who would run both an average case and a worst case through the optimiser in order to see how different the resulting strategies were. If they were seriously sub-optimal for each other's cases, the user would have a choice between wasting some time on most runs of the logic program, or risking disaster on an occasional run. This dilemma is covered further in Section 8.2.

If the set sizes, unification probabilities, and domain sizes supplied to the optimiser are all average case ("expected") values, then it is tempting to hope that the resulting strategy will give the lowest cost averaged over all sets of inputs to the logic program. For this to hold, the estimated costs must be the expected values of the actual costs. However, while the expected value of the sum, difference, or product of independent randomly distributed values is the same function of their expected values, this does not hold for quotients, which are used in estimating the costs. Moreover, it is possible that two values being multiplied will not be independent, and then the expected value can be higher than the product of the expected values of the factors. The consequence of this will be that the true average cost over all runs of the logic program can be higher than the estimated cost of the optimiser's strategy, even if the input data to the program are accurately described by the mean values given to the optimiser. So it may be advisable to re-run the optimiser on "worst-case" values, perhaps some number of standard deviations above the mean, in order to defend against what might go wrong.

Another reason for choosing a strategy under worst-case assumptions might be that the data available for making cost estimates were very uncertain, and it was desired to run the program a few times so as to compare actual costs with the estimates. It would seem to make sense to use a "defensive" strategy on these first few runs, perhaps changing it once better cost data beecame available; this is also mentioned in Chapter 9.

8.2 Controlling Control

As hinted in Chapter 1, any complex program (or other endeavour) can be slowed down by too much control or too much pre-planning just as surely as by too little. In this instance, the control has been designed so as to be efectively free at run-time, but is potentially very expensive at compile-time, and exactly the same consideration applies as when using any optimising compiler: will the time spent in optimisation exceed that saved in execution? In this section, some approaches to avoiding over-optimisation are discussed.

8.2.1 What cannot be done

It is tempting to deal with the question of how much optimisation to use in the same way as we deal with any other optimisation problem, namely estimate how much it will cost versus how much good it will do, and choose a value that gets us the most for the least. Even though the unsolvability of the Halting Problem guarantees that there will be programs whose performance cannot be accurately predicted, this is an approach that has been found to work well in regard to many kinds of program transformation. Most of this dissertation has been concerned with estimating the benefit of a particular program transformation (moving literals around) and deciding how and whether to do it.

Deciding how much work to do on selecting an optimal or near-optimal strategy, and how general a class of strategies to examine, is much harder because there is no known way of predicting the benefits of doing the optimisation. It is not even possible to tell "by inspection" whether a logic program is already optimally arranged, let alone how much improvement is likely to be attainable. The speed of most of the search algorithms described here is also not a well-defined function of the size of the logic program, because we cannot know how many strategies will be of interest to the algorithm, versus the number that will be discarded right away as hopeless. We reluctantly conclude that there is no good way to decide in advance how much optimisation should be done.

8.2.2 Evidence and experience

The decision still has to be made, and can be approached in either of two ways. Experience of past programs can be used to suggest what should be done with this one, or the program itself can be run and observed for evidence on which to base decisions.

The relevant experience will presumably consist of correlations between the sizes of programs, their estimated costs using some default strategy, their estimated and actual costs after being optimised (the actual costs being more important), and, of course, the time taken to optimise them. To use this experience effectively, one would presumably look at the proportional reduction in cost obtained by optimising programs of similar size to the one at hand and the amount of optimiser effort used to achieve this, and then compare the latter number with the estimated cost of the current program.

This approach has a serious weakness in that the "default" strategy, however it is chosen, may be very much better for one program than for another, giving rise to misleading conclusions. There is no escape from this. The size of a program is also a poor guide to its cost; in time, other predictive features may be identified, but good ones are not known yet. So the prospects for using experience of previously optimised programs for guidance to the optimiser seem poor.

However, given that the optimisation in question is being done at compile-time rather than while the program runs, it is plausible that the optimised program will be run many times on various sets of data. We can, therefore, legitimately recommend that some level of optimisation be chosen initially and perhaps altered after a few sample runs, if the execution costs are higher than expected. This approach resembles some of the approaches to run-time control discussed in [Smi85, Ch. 7]. The issues we shall raise here include the choice of the initial level of optimisation and of the criteria for escalating it.

It may be the case that even sample data are not available, or for some other reason it may be impossible to run the program at all. The only way to handle this is to trust that the estimated costs of strategies are accurate, and interleave estimation of the cost of a strategy

with optimisation leading to a better one. Explicit interleaving is not necessary, since the optimiser will produce estimates of the cost of its strategies anyway. This should continue until either the cost of the strategy is comparable to that of the next stage of optimisation contemplated, or the last few attempts at optimisation did not produce enough improvement in the strategy to justify their own cost.

An important parameter of the interleaving is how much optimisation to do before pausing to look at the cost of the strategy obtained: too short an interval of optimisation may fail to improve the current strategy even though improvement is possible, and too long a one may waste a lot of resources before returning with no gain. Certainly the interval should be less than, and preferably small compared with, the estimated cost of the current strategy; there is no reason to make it shorter than, say, the expected error in the estimated cost, since we are already prepared (or should be) to waste such an amount of money.

8.2.3 Budgets and timeouts

There is still the question of what level of optimisation shoud be used initially, before any data whatever are available about how effective the optimisation may be.

We enunciate the following general principle: in the absence of reliable information that any one policy is cheaper than another, use the simplest that shows promise of getting the answer within an acceptable cost. Here "simplest" would mean using the least amount of optimisation. This is based on a belief that the technology for executing logic programs is, and will remain, at a better stage of development than the technology for optimising them, so that it is better to devote resources to running the program than to optimising it, in the absence of firm evidence either way. The proviso about acceptable cost is inserted because there is always some bound on the resources than can or will be devoted to running a program, whether it be imposed by the user's itchy finger on the ABORT key, the policies of the system manager, the MTBF of the hardware, or an objection by interested parties that the problem is inappropriate for machine solution (for an example of this see [Ada79], where

the objectors were eventually pacified by means of an accurate estimate of the running time of the program). For a discussion of what can be done when the amount of resources that can be expended is not known in advance, see [Smi85].

If the system is explicitly told the size of the "budget" for a particular program, it can use this to guide its decisions. The best case is when there are separate budgets for doing optimisation and for running the program once optimised. The only decision that needs to be made then is which method or methods of optimisation are likely to get the best results within the optimisation budget. Even this can, at a price, be ducked by allocating some fraction of the budget to each applicable method, assuming that the budget is big enough to accomodate more than one. If the budget is not very large relative to the cost of estimating the cost of a strategy (which can easily be estimated based on the sizes of the descriptions of R, F, and G plus other parameters as described in Section 3.4) then the quicker optimisation methods should be given priority so that we can be sure of finding some usable strategy before the budget runs out. It is probably a good idea to do this in any case, since the slower methods are unlikely to be harmed by it. For the methods which approach the optimal strategy slowly, there are likely to be no good grounds for believing *a priori* that one of them will do better than another, so they can be run interleaved until the budget is exhausted. Then they will all have had equal opportunities to come up with a strategy. We may want to adjust the interleaving policy according to how much strategy improvement each method is seen to produce per unit of optimisation cost.

The system should make sure that the optimisation returns not only a strategy but also a cost estimate for it, which can be compared with the budget for program execution. If the cost estimate is near or over the budget, the user must be warned, and should be offered the opportunity to increase either of the two budgets. As an aid for this decision, the user deserves to be told something about how much of an improvement the optimiser was able to make relative to the original strategy, and perhaps also about how much better the final

strategy was than those obtained in the middle of the optimisation.

This, of course, is much the same information as the system itself would need if it had only one budget to play with, covering both optimisation and execution. The system's task is harder now, because it may have to choose between using a strategy whose estimated cost (plus the cost of optimisation already done) is barely within the budget, and continuing to optimise in hopes of finding a much better one. In keeping with our political leanings, we shall require that our system refrain from increasing the probability that the budget will be exceeded, based on whatever levels of confidence it may have in the accuracy of its estimates. If necessary, such a system could be modified to incorporate a "price of risk" which would allow it to try optimisations whose benefits were uncertain but potentially large. This is not, however, a dissertation in business administration, and this line of argument will not be pursued further.

8.2.4 Staying within the budget

A major objective must continue to be that easy problems are solved quickly with little overhead. This implies that, before doing any optimisation, the system should estimate the cost of running the program as is (default strategy). We may even go further and require that the cost of such estimation be assessed and, if it is a large fraction of the budget, the estimation be skipped and the program run as is. A rough estimate of how long it would cost to estimate the cost of a strategy can be made very quickly, and users who have problems that both must and can be solved in less time than this should be made to use their own heads.

Once the cost of a strategy has been estimated, any attempt to optimise it must satisfy two conditions. First, in a system obeying the conservative philosophy laid down above, the optimisation should not cost more than the difference between the estimated cost and the budget, in other words the system should not spend money it may need for the execution of the program. And secondly, the optimisation should be expected to cost less than the

strategy itself. This may seem to be a very weak condition. Humans are reluctant to embark on an optimisation whose cost is almost as large as that of the unoptimised program, because of their belief that dramatic improvements to the program are unlikely. But a system that admits frankly to a complete lack of knowledge about the effectiveness of some optimisation does thereby incur an obligation to try it out at least once, provided there is some reasonable hope that it might be useful, because otherwise there will never be any knowledge about its effectiveness.

Chapter 9

Discussion

9.1 Significance

9.1.1 Practical

On the practical plane, the work described here can be regarded as another brick in the great edifice of "programming environments", defined broadly as systems that allow more useful computation to be done per unit of human (and perhaps machine) effort. It is a matter of taste whether the improvement is thought of as stemming from the increased efficiency of the optimised program or from the automation of the optimisation process. One could imagine a naïve programmer who wrote programs without regard for efficiency, in which case use of the optimiser would greatly reduce the programmer's waiting time and the amount of machine work expended, but not the human effort. Or if the programmer were highly sophisticated and had been in the habit of writing efficient programs, the amount of machine effort would not be noticeably reduced, and indeed might be increased, but the programmer could be spared some effort. There would be some overhead cost in either case, namely that of supplying the numbers for the simulator to use in deriving cost estimates. Even this could be largely avoided in a database application, where these numbers would usually be available as part of the description of the database. In expert systems or general logic

programming, it might well be the case that this indirectly domain-dependent information could be appended to the file of rules (or other representation of the logic program) and would require updating only infrequently; it is hard to see how such information could contain a subtle bug requiring many revisions to ferret out. The current implementation of the optimiser tries to read such information from the file in preference to obtaining it interactively.

The accuracy of the optimiser, as compared with a good human programmer, is influenced by two factors. The human presumably understands the problem domain, is aware of the meanings of the predicates involved, and so may be able to estimate costs and do optimisations in ways that are simply not available to the optimiser. But it has (or should have) exact knowledge of the performance of the code used to do the deductions, and it will explore the alternatives far more thoroughly than a human would be likely to. These features may give it an advantage in certain cases. However, no claim is made that the output of the optimiser will necessarily be optimal.

It is, in a sense, not necessary to have accurate information about the problem domain in order to use the optimiser to some advantage. One can envisage feeding different sets of numbers into it in order to see what kind of performance is attainable under different sets of circumstances, and how much the strategy must be changed in order to accommodate to them. There are times when, in the absence of perfect information, one is still desired to "tune" a program, based on whatever bits of guesses or assumptions this will be better than doing nothing. The optimiser could be of assistance in such situations.

9.1.2 Theoretical

in goals, simply by regarding each conjunctive goal as a rule whose output literal was a unit goal. The cases that are solvable in time polynomial in the size of the program description have been identified, as have the reasons why other cases are more difficult. Importantly, features which allow short cuts when they are present in an otherwise difficult case have been characterised.

Secondly, the relationship between local and global optimisations (or ameliorations) of logic programs has been exposed. The majority of prior work has considered one or other of these types, without worrying about interactions between them. Query optimisation in databases has been a local process, and has been kept separate from global or long-term measures such as index generation or reorganisation of relations. Logic program optimisation has concentrated either on improving a single rule amid an unchanging program, or re-arranging a program whose subunits, the rules, are constant. These approaches are now seen to be inadequate in theory, though much easier in practice than fully general optimisation.

Some of the results we have obtained, especially those of Chapters 4 and 6, are applicable over a broader domain than logic programming: they depend mainly on the dependences of one rule on another and on the dichotomy between eager and lazy computation (forward versus backward inference), and so would probably be useful in the optimisation of any computation that was naturally decomposed into smaller ones each of which could be made lazy or eager.

9.2 Alternatives

In this section we discuss some interesting optimisations of logic programs that are related to those we have presented here. They could be attainable by Earl's simple extensions of the [illegible] paradigm.

9.2.1 Caching

Historically, this work grew out of a study of the problem of when to cache facts proved during backward chaining, and which ones to cache. It was eventually decided to shift the emphasis from this to the question of which provable facts should be proved and stored for long-term future use. Selective caching during backward inference is less suitable for this purpose than forward inference with caching of all results, because the facts needed to solve one goal may overlap partly, or not at all, with those needed for another. This raises the spectre of redundant inference.

When a goal is being solved backwards, and it is known that a similar but not identical one was previously solved and its answers cached, it seems obviously better to retrieve those answers from the cache, keeping only those which match the current goal, and then use inference to find the rest. But in general it is difficult to be sure of obtaining all the new answers (those that were not found during the proof of the previous goal) without also allowing the inference mechanism to repeat the deductions that led to the old answers. So in the interests of completeness, backward inference systems normally permit such redundant inference. This can be avoided for subgoals of which it can be shown that all of their solutions are already in the database, in which case it is safe to suppress the inference tree below this subgoal; a particularly easy and common case is where some goals are known to have at most one answer. A full treatment of this is given in [Smi85, Chapter 3].

It is clear that caching, with some extra control such as that proposed by Smith, can be valuable in reducing the cost of inference, especially if it is used judiciously. The cost estimation would become more difficult, and would in fact depend on the order in which goals were asked, since the set of cached answers would depend on the set of goals that had already been asked. This dependence is not easy to eliminate by considering, for example, the set of all answers that are used by more than one goal or subgoal. If a general goal is asked and is followed later by another goal which is more specific than it, the answers to the second goal will be a subset of those to the first, and so no further inference is necessary; if

the order of the goals is reversed, there is no such gain.

To include caching of answers from backward inference, the optimising algorithms would have to be modified to consider three alternatives for each rule (or literal, in the hybrid case), namely forward inference, backward inference with caching, and backward inference without caching. This would inevitably make the optimisation task harder.

9.2.2 Rule splitting and joining

It can be shown that, when optimising a forwards rule, the deductive cost of the rule can sometimes be reduced by splitting it into two or more rules, so that a rule like

$$A_1 \,\&\, A_2 \,\&\, \ldots A_n \Rightarrow B$$

turns into two rules like

$$\begin{aligned} A_1 \,\&\, \ldots A_i &\Rightarrow C \\ C \,\&\, A_{i+1} \,\&\, \ldots A_n &\Rightarrow B \end{aligned}$$

This may, for example, allow duplicate elimination to happen earlier, and it clearly reduces the sizes of the resolvents that must be manipulated.

By contrast, it can also sometimes be useful to reverse this operation, fusing rules together, for either forwards or backwards use. In the forwards case, there is an obvious saving of the costs of storing intermediate facts in the database and retrieving them later; in the backwards case, the expense of looking up rules is avoided. There is a plausible motive for doing such a thing to a logic program: the programmer may have chosen to write short rules for reasons of style or lucidity, even though they are not necessarily the most efficient. There is an even more obvious reason for wanting to split rules up, in the shape of conjunctions of input literals that are repeated across several rules (though this goes beyond what was suggested for a single rule).

With the exception of the common subexpression optimisation, which is already felt to

be fairly well understood, these changes to rules seem to have minor potential for lowering the execution cost.

9.2.3 Symbolic estimates

By analogy with human thinking, and as a palliative for the expected problems of inaccuracy and uncertainty in the numbers which the optimiser requires as its input, it has been proposed that these numbers, and the cost estimates, could be replaced by symbolic expressions or by "fuzzy" quantities. With symbolic expressions, the relative merit of two strategies would reduce to an inequality over the variables representing the set sizes, unification probabilities, and domain sizes; after cancelling out some terms and ignoring others of low order, the truth of such an inequality might reduce to quite a simple question about the relative magnitudes of some values, which could be put to the user. However, even with the linear programming algorithms of Chapter 4, many such comparisons would be made by the linear program, and the processing of expressions for costs would be likely to keep MACSYMA occupied for a long time.

Fuzzy quantities would be easier to work with; probably too easy. If all the numbers fed to the optimiser are drawn from the domain {**zero**, **small**, **medium**, **large**} (or in the case of database applications, {**zero**, **large**, **very large**, **gigantic**}) then there are virtually no grounds on which to distinguish different strategies. This would seem to make the search for an optimal strategy quite easy, since we would only have to find one of a set of strategies whose costs would be indistinguishable. But there would still be no guarantee that the set containing the optimal strategy would be big enough that a member of it would be quickly found, so the cost of search could still be high. There is certainly no obstacle to applying the algorithms given in this dissertation to input numbers that are say rounded to the nearest power of two. However, if the costs of doing the optimisation with definite and fuzzy values are similar, it makes more sense to work with the definite numbers and then blur the result than to abandon definiteness immediately.

9.3 Future Work

There are several directions in which this work must be extended before it can be of use in production logic programming systems, and other directions which would certainly be interesting and profitable.

9.3.1 Recursive rules

Most non-trivial logic programs include rules that are recursive at least in form, even if their true use is more iterative. The cost of using these rules will depend very strongly on non-numeric features of the objects described by the facts – for example, a rule that iterates down lists will take longer if the list is circular. It is important that some means be developed for enabling an optimiser to estimate the cost of using recursive rules. The mechanism for estimation of costs that has been described in Chapter 3 is crude and unsatisfactory enough for non-recursive rules, though attempts are being made [Dem80,Chu83,PC84] to improve on the ideas underlying it; for recursive rules, it is quite inadequate.

9.3.2 Adaptive control strategies

As was hinted in Section 8.2, experience gained by running a logic program may be useful in deciding how much work should be done on optimising it. This also holds true at a lower level: such experience may, and probably can, be helpful to the optimiser itself in predicting the costs of strategies. This observation leads us in two directions. On the one hand, information gathered during one or more runs of a program may enable cost estimates to be made more accurate, so that a better strategy can be found for subsequent runs. On the other hand, during a single run of the program it may be useful to compare the observed costs with those predicted for the strategy in use, and if there are deviations, especially in the upwards direction, the strategy may have to be changed. Since these changes must be decided upon at run time, the experts or optimisation algorithms at hand must

be suitable for this. There are probably interesting issues to be explored in the design of strategies that make provision for subsequent alteration based on observed behaviour.

9.3.3 Compilation

Large increases in the speed of logic program execution are available through compilation into lower level languages (see, for example, [War77]). Depending on the details of the compiler used, there may or may not be difficulties in estimating the cost of running the compiled code. Still, we expect that it will have to perform much the same deductive operations as would be done by an interpreter, and so the cost will change only by a constant factor. Aside from the obvious need to tell the compiler which direction has been chosen for each rule, it may also be that the compiler can generate more efficient code for a backwards rule if it knows something about the set of goals likely to be solved by the rule. For example, if there will be no goals in which a certain variable is free, the code generated for pattern matching could be adjusted to assume that it will be bound. Or if the goal set is expected to be large, the compiler might choose to generate bulkier but more time-efficient code than would be the case for a small goal set. Such information will be available as a by product of the strategy optimisation, and ought to be passed on to the compiler.

9.3.4 Hot spots

In the same vein, but rather more speculatively, the cost estimates could be used to find "hot spots", parts of the program where large costs are incurred, corresponding to inner loops in more conventional programming languages. These could be returned to the programmer for hand optimisation, or extra effort could be expended on optimising them automatically. For example, a full exhaustive search of the possible changes to the strategy in the vicinity of such a hot spot could be done, instead of one of the abbreviated or randomised searches mentioned in earlier chapters. This is likely to pose some problems, because a rule may cause a large change to the cost of a program without having a prominently high local cost.

if it generates moderate numbers of subgoals to rules that do have high expense per subgoal. Pinning the blame for a high estimated cost where it belongs, or localising a hot spot, can thus be difficult.

Even poor detection of hot spots will be useful for quite a different reason, namely the speed of the search algorithms given in this dissertation. It has been pointed out in several places that the search will approach the optimal strategy more rapidly, and can perhaps be terminated sooner at a satisfactory strategy, if the rules whose directions have the largest effects on the uncertainty in the total cost are added early to a partial strategy. These will frequently, though not always, be the rules whose contributions to the total cost are large (whether by their own computational cost or by the work they cause other rules to do). If we can identify even a few of these rules, the search algorithms will probably benefit by considering them first.

9.3.5 Reformulation

The true cause of inefficiency in a logic program may not be the arrangement of the terms in its rules, but rather the choice of vocabulary in which to describe the problem being solved; psychological studies have found that good human problem solvers spend significantly greater proportions of their total problem-solving effort on deciding how to "encode" a problem than do poor problem solvers. This was recognised a long time ago by Amarel [Ama68]. It is doubtful whether the methods presented in this dissertation will be of any use in finding ways to improve the representation of a problem in predicate calculus form, but they should be of use in determining whether a particular representation can be run faster than another. If a new representation is found which improves on the old one, but the facts (and perhaps goals) are still available only in the old representation, then the question of whether to translate them all into the new one before starting to solve the problem, or delay translation of each proposition until needed, is precisely a choice between forwards and backwards inference.

A possibly easier kind of reformulation that could be contemplated within the framework built up in this dissertation would be finding and separating out sets of input literals that occur together in several rules. Such a set may define a useful predicate, and there may be some benefit available by writing a separate rule to deduce instances of it, especially if the rule be used forwards. In a sense this is no more than a wider-ranging version of the rule splitting mentioned above.

9.3.6 Parallel Execution

It has been implicitly assumed that all the deductions will be done serially on a single processor. While deduction has inherently serial aspects (there are reasons why logic programmers speak of forward and backward *chaining*), it can also frequently be speeded up by processing different parts of a program in parallel with each other. This is largely orthogonal to the speed-up available by reducing the number of elementary deductive operations required to solve a problem, which is what we have focussed on. However, when a set of processors is dedicated to solving a problem in parallel, and only imperfect parallelism can be achieved, so that some of the processors are idle some of the time, this idle time should perhaps be added to the cost of solving the problem, since it represents CPU cycles that may not be available for any other purpose. Thus the estimated cost of a strategy may change in complicated ways under parallel processing, depending on whether (for example) forward inference turns out to be more readily parallelisable than backward inference. This will probably have severe consequences for those optimisation methods that make use of estimates for the cost of a single rule or the expense of a single goal. It is already known that literal ordering may be different, since it can be cheaper to solve several input literals of a rule in parallel and then combine the sets of answers than to work sequentially. Cost estimation itself may become harder if the simulator is required to simulate the task allocation algorithm of a parallel architecture.

Bibliography

[Ada79] Douglas Adams. *The Hitchhiker's Guide to the Galaxy.* Pan Books, 1979.

[Ama68] S. Amarel. On representations of problems of reasoning about actions. In Michie D., editor, *Machine Intelligence 3*, pages 131–171, Edinburgh University Press, Edinburgh, 1968.

[ASK80] M. Astrahan, M. Schkolnick, and W. Kim. Performance of the System R access path selection mechanism. In *Information Processing 80*, pages 487–491, IFIP, 1980.

[Ast76] M. Astrahan *et al.* Sytem R: relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.

[AvL85] E.H.L. Aarts and P.J.M. van Laarhoven. Statistical cooling: a general approach to combinatorial optimization problems. *Philips J. Res.*, 40:193–226, 1985.

[Boy71] Robert S. Boyer. *Locking: A Restriction of Resolution.* PhD thesis, University of Texas at Austin, August 1971.

[Cha81] D. Chamberlin *et al.* A history and evaluation of System R. *Communications of the ACM*, 24(10):632–646, October 1981.

[Chu83] C.W. Chung. *A Query Optimisation in Distributed Database Systems.* Technical Report CRL-TR-4-83, University of Michigan, 1983.

[CL73] C.L. Chang and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[CM77] Ashok Chandra and Philip Merlin. Optimal implementation of conjunctive queries in relational databases. In *Ninth ACM Symposium on the Theory of Computing*, pages 77–90, Association for Computing Machinery, 1977.

[CM84] William F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Com78] Douglas Comer. The difficulty of optimum index selection. *ACM Transactions on Database Systems*, 3(4):440–445, December 1978.

[Dan65] George B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1965.

[Dem80] R. Demolombe. Estimation of the number of tuples satisfying a query expressed in predicate calculus language. In *Proceedings of the 6th VLDB Conference*, pages 55–63, IEEE, 1980.

[FG85] J. J. Finger and Michael R. Genesereth. *RESIDUE - A Deductive Approach to Design Synthesis.* Knowledge Systems Laboratory Report KSL-85-1, Stanford University, January 1985.

[FST86] S. Finkelstein, M. Schkolnick, and P. Tiberio. *Physical Database Design for Relational Databases.* Technical Report RJ5034, IBM Almaden Research Center, February 1986. Submitted to ACM TODS.

[GA83] Malik Ghallab and Dennis Allard. A_ϵ - an efficient near admissible heuristic search algorithm. In *Proceedings of the Eighth International Joint Conference*

on Artificial Intelligence, pages 789–791, International Joint Conference on Artificial Intelligence, 1983.

[Gen82] Michael R. Genesereth. *An Introduction to MRS for AI Experts.* Knowledge Systems Laboratory Report KSL-82-27, Stanford University, November 1982.

[Gen84] Michael R. Genesereth. *Partial Programs.* Knowledge Systems Laboratory Report KSL-84-1, Stanford University, November 1984. Submitted to JACM.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, 1979.

[GM78] H. Gallaire and J. Minker. *Logic and Data Bases.* Plenum Press, New York, 1978.

[GM80] John Grant and Jack Minker. Optimization in deductive and conventional relational database systems. In Herve Gallaire, Jack Minker, and Jean Marie Nicolas, editors, *Advances in Database Theory*, pages 195–234. Plenum Press, 1980.

[GMN80] Herve Gallaire, Jack Minker, and Jean Marie Nicolas. *Advances in Database Theory.* Volume 1, Plenum Press, New York, 1980.

[Hal76] P.A.V. Hall. Optimisation of a single expression in a relational data base system. *IBM Journal of Research and Development*, 24:4:244–257, 1976.

[Hew75] C. Hewitt. How to use what you know. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, page 189. International Joint Conference on Artificial Intelligence, 1975.

[HNS] [illegible]

Hol62 John H. Holland. Outline of a logical theory of adaptive systems. *Journal of the ACM*, 9(3):297–314, 1962.

Hol75 John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

IK84 T. Ibaraki and T. Kameda. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

Jar85 M. Jarke. Common subexpression isolation on multiple query optimization. In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, pages 191–205, Springer-Verlag, 1985.

JK84 M. Jarke and J. Koch. Query optimization in relational databases. *ACM Computing Surveys*, 16(2):111–152, June 1984.

KBZ86 Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. *Efficient Optimization of Nonrecursive Queries*. Technical Report, Microelectronics and Computer Technology Corporation, February 1986.

KGV82 S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. *Optimization by Simulated Annealing*. Technical Report RC9355, IBM Thomas J. Watson Research Center, April 1982.

[illegible] [illegible] optimization of relational queries: a first step. In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, pages 206–226, Springer Verlag, 1985.

[illegible] *Computer Programming*. Volume 1, Addison-[illegible]

[Kor85] Richard Korf. Iterative-deepening A*: an optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1034–1036, International Joint Conference on Artificial Intelligence, 1985.

[Kow75] R. Kowalski. A proof procedure using connection graphs. *Journal of the ACM* 22(4):572–595, October 1975.

[Mau84] Michael L. Mauldin. Maintaining diversity in genetic search. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 247–250. American Association for Artificial Intelligence, 1984.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MS81] D. P. McKay and S. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 368–374, Vancouver, August 1981.

[NH80] S. A. Naqvi and L. J. Henschen. Performing inferences over recursive data bases. In *Proceedings of the First National Conference on Artificial Intelligence*, pages 263–265, American Association for Artificial Intelligence, August 1980.

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, 1980.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by co-operating decision procedures. *ACM Transaction on Programming Languages and Systems*, 1(2):245–257, October 1979.

[Ove] Ross Overbeek. Complexity and related enhancements for automated theorem proving programs. *Computing and Mathematics with Applications*, 2(1):1–16,

[Rou65] [illegible]

Journal of the [illegible]

[Rou82a] Nicholas Roussopoulos. Indexing views in a relational database. *ACM Transactions on Database Systems* [illegible] 258–290, June 1982.

[Rou82b] Nicholas Roussopoulos. The logical access path [illegible] *Transactions [illegible] Software Engineering* [illegible]

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data*, pages 23–34. Association for Computing Machinery, 1979.

[illegible] Optimizing the performance of a relational algebra [illegible] *Communications of the ACM*, 18(10):568–579, October [illegible]

[illegible] David E. Smith and Michael R. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 26(2):[illegible], 1985.

[illegible] Robert Shostak. Deciding combinations of theories. *Journal of the ACM*, [illegible], 1984.

[illegible] D. Sleator. An O(nm log n) *Algorithm for Maximal Network Flow*. PhD thesis, Stanford University, November 1980.

[Sma83] Steve Smale. On the average number of steps of the simplex method of linear programming. *Mathematical Programming*, 27(3):241–262, December 1983.

[Smi85] David E. Smith. *Controlling Inference*. PhD thesis, Stanford University, August 1985.

[SS81] M. Schkolnick and P. Sorenson. *The Effects of Denormalization on Database Performance*. Technical Report RJ3082, IBM San Jose Research Laboratory, December 1981.

[Sti85] Mark Stickel and Mabry Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1073–1075, International Joint Conference on Artificial Intelligence, 1985.

[Sua86] Roberto Suaya. 1986. Private communication.

[SVG79] *Stanford Pascal Verifier User Manual*. March 1979. STAN-CS-79-731.

[Ull82] Jeffrey D. Ullman. *Principles of DataBase Systems*. Computer Science Press, Rockville Maryland, 1982.

[Ull84] Jeffrey D. Ullman. *Implementation of Logical Query Languages for Databases.* Technical Report STAN-CS-84-1000, Stanford University, May 1984.

[War77] D. H. D. Warren. *Implementing Prolog - Compiling Predicate Logic Programs.* D.A.I. Research Report 39 and 40, University of Edinburgh, May 1977.

[War81] D.H.D. Warren. Efficient processing of interactive relational database queries expressed in logic. In *Proceedings of the Seventh VLDB conference*, pages 272–281, IEEE, 1981.

[WOLB84] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications.* Prentice-Hall, 1984.

[WY76] E. Wong and K. Youssefi. Decomposition - a strategy for query processing. *ACM Transactions on Database Systems*, 1(3):223–241, September 1976.

[Yao79] S.B. Yao. Optimization of query evaluation algorithms. *ACM Transactions on Database Systems*, 4(2):133–155, June 1979.

END

5-87

DTIC